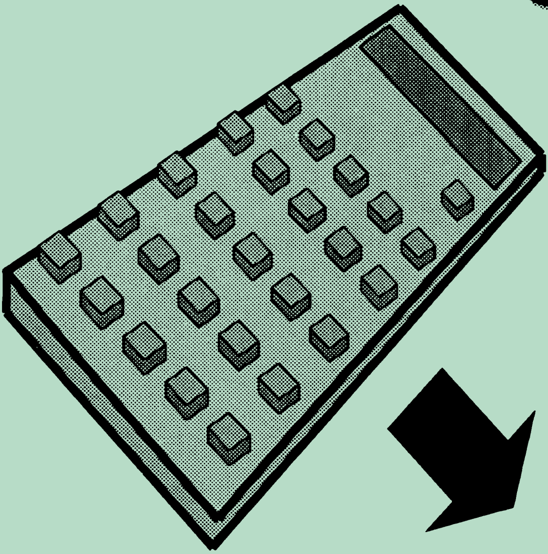
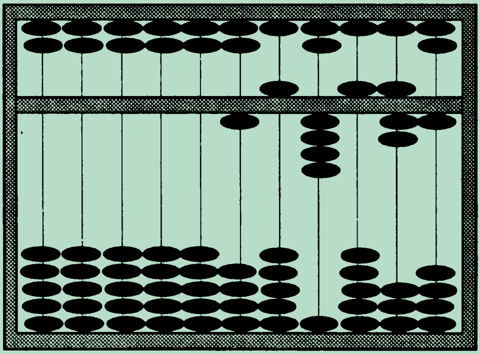
Assembler gjort Simpel





KR. 75,00

Af Klaus-Henrik Dalkov

HiScore Professionel, Til dig der <u>vil</u> noget med din computer!
I HiScore Professionel tibyder vi dig:

❍ Feature-artikler om det der rører sig i den spændende og forunderlige verden omkring computeren. Pirater, produktion af CD-ROM, bag om Jurassic Park, SEALS - hårdkogte soldater indtager computeren og meget mere. Kun i HiScore Professionel.

❍ Danmarks bedste guide til hvilke spil der er pengene værd, og hvilke du skal holde dig langt fra. Altid test af de nyeste spil

❍ Bliv bedre til at *compute*, med HiScore Professionels 'Tips og Tricks' sektion. Hver måned spækfyldt med hjælp til netop dit yndlingsspil. Giver dig mulighed for at gennemføre det, før din nabo.

❍ Alle de bedste tilbud fra forhandlerne finder du i HiScore Professionel.

Et abonnement giver dig mulighed for at modtage bladet ind af brevsprækken dugfrisk fra trykkeriet. Ikke noget med at stå i kø, lede efter kiosker der ikke har udsolgt o.s.v. HiScore Professionel er dit blad direkte til dig.

Indbetaling foregår på posthuset , hvor du tager et girokort og indsender et beløb, svarende til din bestilling på giro: 393 04 08. Adresseres til Dansk Medie Hus ApS.
1/1-årsabonnement: 355,-
1/2-årsabonnement: 170,-

Denne bog giver dig et godt førstehåndsindtryk af computersproget asembler.

En varm tak for korrekturlæsning til
mine gode venner
Mark Kønigsfeldt
og
Jesper Strømstad

Indhold

Indledning	3
Hvorfor assembler	4
Hvilken assembler	5
Computerens virkemåde generelt	6
Talsystemer	8
Introduktion til hukommelse	13
Assemblersprogets format	14
Registre	15
Syntaxen i denne bog	17
Assembler direktiver	18
Programafslutning	21
Dataflytning	22
Adressering	26
Forgreninger og hop	29
Sammenligninger	31
Simpel Aritmetik	33
Løkker	37
Subrutiner	39
Bit kommandoer	41
Bit shifting og rulning	43
Logiske Operationer	46
Et par facts om CLI	51

Assembler gjort Simpel:
Udgiver: Dansk Medie Hus ApS - Forlag & Marketing
Forfatter: Klaus-Henrik Dalkov
Layout: Christian G. Larsen
Trykt: Bakager Tryk

Bogen kan købes hos din lokale forhandler eller ved Dansk Medie Hus ApS på følgende girokonto: 393 0408.

Dansk Medie Hus ApS
Hillerødgade 81 - 83
2200 København N.

Indledning

Hvis du har købt denne bog, har du vel besluttet dig til at kaste dig ud i assembler-programmeringens kunst, men inden vi går i gang, må vi hellere få et par sandheder frem. Assembler er i bund og grund et andet ord for 'symbolsk maskinkode', altså et sprog der kommer uhyggeligt tæt på den kode, computeren forstår. Det siger sig selv, at det dermed også er en af de mest avancerede programmerigsformer, man kan kaste sig ud i, og jeg er bange for, at det vil være en halsløs gerning, hvis du er totalt begynder inden for programmering.
Kender du lidt til et højniveau-sprog, som f.eks. BASIC, vil man være godt hjulpet, men det kræver en vis 'omstilling' at skifte til assembler, og det er her, denne bog tager sit udgangspunkt. Men vær opmærksom på, at det nok kommer til at koste sved, tårer og, først og fremmest, tid!
Er du bekendt med C, er en hurtig gennemlæsning af denne bog muligvis nok til at sætte dig i gang med assembler, men husk på at du pludselig skal til at fremstille alle rutiner selv. Ydermere er det nemt at blande C og assembler i samme program, så du kan nøjes med at lave de mest tids-kritiske rutiner i assembler.

Denne bog består af to dele, en teoretisk del og en mere praktisk betonet. Jeg anbefaler, at den teoretiske del bliver læst igennem først, inden du giver dig i kast med selve instruktionssættet, også selvom noget af teorien er lidt svært forståeligt. Efterhånden som du læser den praktiske del, vil du nok få brug for at repetere nogle af de teoretiske afsnit, men det har været nødvendigt at opbygge bogen på denne måde, da der er så meget der skal læres og så få sider at gøre det på.

Til sidst vil jeg lige sige, at jeg ikke gennemgår hele instruktionssættet i denne bog. Det er der simpelthen ikke plads til! Supplerende litteratur bliver derfor nødvendig, hvis du vil gå videre med assembler-programmeringen.

Hvorfor assembler?

Ja, hvorfor vælger nogen at programmere i assembler? Det er et af de sprog, der er sværest at lære, har den længste program-udviklingsstid, har den længste debugningstid, og er kun i ringe omfang portabel mellem processorer. Men der er et par enkelte fordele, der vejer meget tungt:

1) Man har nærmest total kontrol over computeren
2) Program-udførelsestiden er nærmest optimal

Dette gør assembler meget velegnet til spil og demoer, hvor al den kraft, der kan vrides ud af maskinen bliver udnyttet. Til gengæld er det ikke altid umagen værd at lave system-programmering i assembler, selvom visse tåber (forfatteren iberegnet) gør det alligevel.

> ” *Hvis noget kan gå galt, vil det gå galt*
>
> *- Murphy's 1. lov*

Efter disse få ord håber jeg, du er blevet klar over hvad du er i gang med at kaste dig ud i, og samtidig har mod til at fortsætte.

Problem	**BASIC**	**C**	**Assembler**
Udviklingstid	Lav	Rimelig	Lang
Debugning	Lidt	En del	Meget
Portabilitet	God	God	Ringe
Anskaffelsespris	Rimelig✔	Dyr✗	Rimelig✗
Dansk litteratur	Meget	Få	Få
Systemkontrol	Ringe	God	Optimal
Udførelsestid	Elendig	God	Meget god

✔ Gratis på WB 1.3
✗ Findes dog også som Public Domain

Som skemaet angiver, er udviklingstiden, og bagefter debugningstiden, væsentligt længere end ved højniveau-sprog. Dette skyldes til dels, at der i maskinkode ikke bliver spildt tid på at checke om den rigtige datatype er brugt, eller om man overskrider et array. Vil man spilde tid på at checke dette, er det op til programmøren selv at gøre det. Dette vil utvivlsomt give nogle Guru'er under udviklingsarbejdet, men det er også grundlaget for den ekstremt høje udførelseshastighed bagefter.

For at kunne programmere i assembler må man have et program, der kan oversætte source-koden til maskinkode. Dette program kaldes for forvirringens skyld en Assembler. Der findes temmelig mange på markedet, men jeg vil kun nævne nogle få.

A68k

er en Public Domain (altså gratis) Assembler, der kan det mest nødvendige for begynderen, men vil man gå videre med programmeringen, er denne Assembler nok ikke fremtiden.

AsmOne

Utrolig god, danskudviklet Assembler, med indbygget source-level debugning og et væld af funktioner og muligheder. Er muligvis den bedste Assembler til programmering af spil og demoer, mens den har lidt bøvl med nogle af Commodore's Include-filer til systemet.

DevPac

Utrolig god, professionel Assembler med mange muligheder og medfølgende debugger. Snupper alle Commodore's Includes uden at kny, men den er ret dyr.

Valget er dit, men vil du fortsætte med assembler i lang tid, gør du dig selv en tjeneste ved at købe én af de to sidstnævnte. AsmOne er den nemmeste at gå til (efter min mening), men DevPac i version 3 er den mest professionelle.

Har du allerede en C-Compiler med tilhørende Assembler, f.eks. **SAS** eller **Aztec**, kan du sagtens benytte denne, men vær opmærksom på, at de kun er lavet til at kombinere assembler og C. Jeg kender ikke meget til Aztecs assembler-del, men der er nogle enkelte bugs (fejl) i SAS's, disse kan man dog forholdsvis nemt komme udenom.

Brugen af de enkelte Assemblere må du læse om i manualerne, ligeledes vil begrebet 'Linkning' forblive ubeskrevet i denne bog. Hvis du vælger at benytte A68k får du, på ét eller andet tidspunkt, brug for at linke - højst sandsynligt med programmet 'BLink', hvor imod linkning sker automatisk i AsmOne og DevPac.

Denne bog vil stige betydeligt i værdi, hvis du vælger en Assembler, hvor man kan se indholdet af sine registre m.v. efter kørsel af et program. Her vinder Asmone i kapløbet blandt de nævnte Assemblere foran DevPac, og A68k halter noget bagud. Alt i alt kan det altså betale sig at gå den kommercielle vej.

Computerens virkemåde generelt

Vores computere er opbygget som et digitalt elektronisk kredsløb. At det er digitalt vil sige, at computeren i princippet kun skelner mellem to tilstande i signalbehandlingen, nemlig om der løber en strøm, eller der ikke gør.
Som det tekniske modstykke til mennesket, er en computer bygget op om en central hjerne, som kaldes en CPU (Central Processing Unit). Der er dog flere CPU'er i nogle maskiner, men ikke i de eksisterende Amiga-modeller, så hvad bryder vi os om det? Hjernen styrer så sine organer efter behov, som selvfølgelig er forskellige elektroniske hjælpe-kredsløb, og meget menneskeligt har hjernen også en hukommelse, som glemmer alt, hvis den ikke får energi. Hjernen i Amiga'erne er fra Motorola's MC680x0 familie, og alt efter hvilken maskine man har, vil det være en MC68000, MC68020, MC68030 eller MC68040. MC68010 er ikke benyttet som standard, men nogle folk elsker at bygge en sådan ind i deres gamle Amiga for hastighedens skyld.

Ligesom et spædbarn har computeren dog nogle simple medfødte reflekser, som den ikke glemmer, selvom strømmen slukkes. Disse informationer ligger i en hukommelsestype, man kalder ROM (Read Only Memory), altså en hukommelse man kun kan læse fra og ikke tilføje nye ting til. Selve hukommelsen, hvor du kan lære maskinen forskellige ting (ved at indlæse et program), kaldes RAM (Random Access Memory), hvilket er en hukommelse, du både kan skrive til og læse fra. RAM'en mister sit indhold når strømmen slukkes.

> ” *Derpå sagde Gud: »Lad os gøre mennesker i vort billede, så de ligner os, til at herske over havets fisk og himmelens fugle, kvæget og alle vildtlevende dyr på jorden [...]«*
>
> *- 1. Mosebog (1:26)*

Hukommelsen består af en utroligt masse celler, kaldet bits, som kan have en digital værdi, altså 'strøm' eller 'ikke strøm' som gerne oversættes til tal 1 eller 0. Computeren forstår som sagt kun bit-værdierne, så det er altså variationen af 1'er og 0'er som fortæller CPU'en hvad den skal lave. Nu vil det være helt uoverskueligt at betragte alle bits individuelt, så nogle smarte mennesker har fundet på nogle navne til forskellige længder af bitstrenge:

Længde af streng	Navn
1	Bit
4	Nibble el. Nybble
8	Byte
16	Word
32	Long word

Da hukommelseskredsløbene stort set består af én lang, uoverskuelig bit-streng, har man gjort det samme, som man gør ude i den vilde verden. Først propper man 8 bits sammen i en familie, som altså kaldes en byte, og hver familie får en adresse på vejen, således at første adresse hedder 0, den næste hedder 1 osv. Laver man et dobbelt-hus med to familier, et word, er der dog den begrænsning, at et word kun kan starte på et lige adresse-nummer, altså 0, 2, 4, 6 osv. Det samme gælder for 4-familieshuse, long words, der i princippet dog vil (og i nogle tilfælde skal) ligge på adresserne 0, 4, 8, 12 osv., men det er normalt ingen betingelse - alle lige adresser er gyldige.

> *Og Gud skabte mennesket i sit billede; i Guds billede skabte han det, [...]*
>
> *- 1. Mosebog (1:27)*

Lad det være op til dig selv at bedømme, hvorvidt mennesket nu selv har leget Gud. Gudskelov har vi endnu ikke fundet Visdommens Tekniske Æble, så et nyt syndefald synes endnu ikke at stå for døren.

Talsystemer

Dette kapitel omhandler nok det mest kedelige, en begynder skal lære, men det er en nødvendighed for at lette tilgangen til maskinkode. Kender du det binære- og det hexadecimale talsystem i forvejen, kan du roligt springe videre, men er du det mindste i tvivl, bør du repetere din viden med dette kapitel. Det er essentiel viden, og du slipper ikke uden om det i resten af bogen.

> *Den eksakte matematiske formel for Murphys Lov på computerområdet lyder: 1 + 1 = 2, hvor '=' er et symbol med betydningen 'sjældent, hvis overhovedet'.*
>
> *- fra Murphy's Computer-love*

Vi kender allesammen det decimale talsystem, 10-talssystemet, som vi nærmest er blevet opdraget med, men vi repeterer lige 'virkemåden' af cifrenes placering i et decimaltal. Vi kalder det for 10-talssystemet fordi det beskriver 10 forskellige cifre, som kan sættes sammen til større tal. Med andre ord er grundtallet 10, som også kaldes 'basen'. De enkelte cifre i et tal kalder vi 'tællere', fordi de optæller hvor mange enere, tiere osv., hver plads i tallet beskriver, og cifferets plads kan beskrives som en eksponent til basen. Lad os tage et eksempel:

Eksempel 1: 245 indeholder 5 enere, 4 tiere og 2 hundreder, eller mere skematisk:

Decimaltal	Tæller		Base	Eksponent	Del-resultat
245	5	*	10	^0	$5 \cdot 10^0 = 5$
	4	*	10	^1	$4 \cdot 10^1 = 40$
	2	*	10	^2	$2 \cdot 10^2 = 200$
Sum					**245**

Som du ved, har man vedtaget i matematiske kredse, at alle tal man opløfter til 0'te potens giver 1. Der er en anden ting, man bør bemærke sig, og det er at alle tal læses fra højre mod venstre, altså mod læseretningen, og dermed er det ciffer, der står længst til venstre altid MEST betydende, da det giver det største bidrag, og det ciffer længst til højre er MINDST betydende, da dets bidrag er det mindste af alle cifres. Lad os så betragte et lille gange-stykke, f.eks. 87•93. Alle os danskere, der har taget, i hvert fald det meste af folkeskolen, ville lave en lille opstilling i stil med (***Fig1 A***)

Fig1, A,B,C: Multiplikation på dansk og russisk

A/	B/		C/
87•93	87	93	93
651	43	186	186
7440	21	372	372
8091	10	744	~~744~~
	5	1488	1488
	2	2976	~~2976~~
	1	5952	5952
			8091

Og så er det jo ikke så svært at gøre, men hvad ville en bonde fra en fjern sibirisk bjergegn mon gøre. Han har ikke gået i folkeskole, og kan ikke en gang regne med brøker og decimaler. Jo se, Ivan ville benytte det, man lidt slang-agtigt kalder 'russisk multiplikation' eller 'halvering og fordobling'. Det går ud på at man laver to simple tal-rækker, hvor man bliver ved med at halvere den første, indtil der ikke længere er noget at halvere, og samtidig fordobler man den anden et tilsvarende antal gange. Er halvdelen af et tal ikke helt, men giver en rest, smider man bare resten væk (f.eks. er halvdelen af 25 = 12) (***Figl B***).

Når Ivan er nået så langt, leder han den første tal-række igennem efter lige tal, og sletter det tilsvarende tal i anden række. Han finder 10 og 2, dem sletter han sammen 744 og 2976, og så lægger han de resterende tal i anden række sammen (***Figl C***).

Det er muligvis lidt af en omvej, men når Ivans manglende kendskaber tages i betragtning, er metoden faktisk temmelig genial, og han ville nok stå og grine fjoget eller uforstående, hvis man fortalte ham, at han netop har benyttet det binære talsystem.

Så langt så godt. Problemet er, at digitale computere, som Amiga'en, kun kender to forskellige tilstande, 'strøm' eller 'ikke strøm', og dette omsættes smartest med et talsystem, der kun har 2 forskellige cifre, altså 2-talssystemet. I dette system kan vores tæller altså kun være 0 eller 1, så vores base er nu 2, og de enkelte cifre i det binære tal kan direkte opfattes som bits i en computer. Lad os tage et eksempel:

Eksempel 2

Binært tal	Tæller		Base	Eksponent	Del-resultat
%11110101	1	*	2	^0	$1 \cdot 2^0 = 1$
	0	*	2	^1	$0 \cdot 2^1 = 0$
	1	*	2	^2	$1 \cdot 2^2 = 4$
	0	*	2	^3	$0 \cdot 2^3 = 0$
	1	*	2	^4	$1 \cdot 2^4 = 16$
	1	*	2	^5	$1 \cdot 2^5 = 32$
	1	*	2	^6	$1 \cdot 2^6 = 64$
	1	*	2	^7	$1 \cdot 2^7 = 128$
Sum					245

Læg mærke til at vi sætter et % foran binære tal, så vi kan skelne dem fra andre talsystemer. Og igen læser vi tallet fra højre mod venstre.
Vi ser altså, at det binære tal %11110101 er lig med det decimale tal 245. Læg iøvrigt mærke til talrækken, man får frem ved de binære tal: 1, 2, 4, 8, 16, 32 osv., allesammen multipla af 2 (basen). Du vil sikkert støde på denne række ofte, da de fungerer som de 'runde' tal i 2-talssystemet, og svarer til 1, 10, 100, 1000 osv. i 10-talssystemet.

Talsystemer

Lad os vende tilbage til Ivans tal-rækker, men denne gang halverer vi begge rækker, og i stedet for at overstrege nogen tal, skriver vi i stedet 1 ud for de ulige tal og 0 ud for de lige (***Fig2A***).

Fig 2, A,B: Binær omskrivning af decimaltal

A/	87	1	93	1		%1010111 = 87
	43	1	46	0		%1011101 = 93
	21	1	23	1		
	10	0	11	1		
	5	1	5	1		
	2	0	2	0		
	1	1	1	1		

Nu er Ivan stolt over sit resultat, men han har ikke den fjerneste ide om, hvad han har udrettet. Men det har du selvfølgelig, for hvis du læser dine 0'er og 1'er-rækker nedefra, altså (***Fig2B***), har du fundet de binære udtryk for de to tal!

Som du kan se, bliver et binært tal meget hurtigt hulens langt, så vi må nok finde på en smartere måde at skrive dem på.
I sidste kapitel så vi, at en nibble er den korteste bitstreng, der har et navn. Derfor er det måske smart at finde på et talsystem, der netop kan skrive en nibble med kun ét ciffer.
Ser vi på et binært tal med 4 cifre, er det største tal vi kan skrive **%1111 = 15.**

Det eneste talsystem, der har 15 som højeste eksponent, er 16-talssystemet, også kaldet det hexadecimale talsystem. Dette system fungerer på nøjagtig samme måde som de to andre systemer, det indeholder bare 16 forskellige tal i stedet. Dette kunne selvfølgelig være et problem, da vi med vores arabiske tal kun har cifrene 0-9, men så må vi låne et par bogstaver til at fylde ud med:

$A = 10, $B = 11, $C = 12, $D = 13, $E = 14, $F = 15

Som du kan se, planter vi et $ tegn foran hexadecimale tal, igen for ikke at rode rundt i de forskellige talsystemer. Og lad os så tage et eksempel:

Eksempel 3

Hexadecimalt tal	Tæller		Base	Eksponent	Del-resultat
$F5	5	*	16	^0	$5 \cdot 16^0 = 5$
	F	*	16	^1	$F \cdot 16^1 = 240$
Sum					**245**

Vi har altså nu at 245 = %11110101 = $F5.

Hvis du øver dig lidt med konverteringen af talsystemerne til det decimale talsystem, bliver det efterhånden en vanesag, omend den altid er lidt besværlig. Gudskelov findes der et utal af PD programmer, der kan gøre det for dig - der følger endda et lille program med på WB1.3 disketten, der hedder 'Eval', som foruden omregning af talsystemer også har nogle logiske operationer og bitkommandoer (mere om disse senere).
Der er dog én af konverteringerne, der er nem at have med at gøre, nemlig mellem det binære og det hexadecimale system. Det skyldes netop, som nævnt tidligere, at ét hex-ciffer altid kan skrives med præcis 4 binære cifre. Har man en bitstreng, kan man derfor bare splitte den op i fire cifre ad gangen (fra højre mod venstre) og oversætte delene hver for sig; tid til et eksempel:

Eksempel 4: Konvertering mellem binære- og hex-tal

```
%1111  0101     opdel det binære tal i blokke af fire (start fra højre)
$  F      5     idet %0101=1+4=5 og %1111=1+2+4+8=15=F
```

Det binære tal kan være så langt du måtte ønske det, og er der ikke et helt antal af fire cifre, fylder du bare ud med nuller forrest:

```
%1  0101 = %0001  0101 = $15
```

Selvfølgelig kan man lige så let konvertere den anden vej.

Negative tal

Vi skal afslutte læren om talsystemer med det absolut værste, du kommer ud for i denne bog: De negative tal.
Computeren kan ikke sådan bare sætte nogle fortegn rundt omkring, men den har et flag, den kan hejse og vinke med, hvis den støder på et negativt tal. Dette kommer vi tilbage til senere.

Hvis du har snuset rundt i Commodore's Include filer, har du muligvis bemærket nogle underlige måder at beskrive længden af et tal på, jeg mener, BYTE, WORD og LONG er til at forstå, men UBYTE, UWORD og ULONG ligner mest af alt stavefejl. Der er faktisk en logisk forklaring på det mystiske U; det betyder 'Unsigned' hvilket kan oversættes med 'uden fortegn'. Hvis vi regner med fortegn, kalder vi det 'Signed', men der er ikke grund til at skrive noget S - hvis U'et ikke er der, er tallet med fortegn.

Ser vi på et word (16 bits) kan det beskrive $2^{16} = 65536$ forskellige tal, eller skrevet lidt anderledes: **0 <= UWORD <= 65535,** altså alle tal mellem 0 og 65535 begge inclusive.

Talsystemer

Når processoren vil skelne mellem et positivt og et negativt tal, ser den på den øverste bit (bit 7 i en byte, bit 15 i et word, bit 31 i et long word), der kaldes MSB; Most Significant Bit eller 'mest betydende bit'. Hvis MSB er 0, er tallet positivt, og hvis MSB er 1, er tallet negativt. Ser vi på et word, kan vi altså ikke længere bruge alle 16 bit til at beskrive tallet, men kun de 15 af dem. Det giver $2^{15} = 32768$ forskellige positive tal og lige så mange negative:
-32768 <= WORD <= 32767, læg mærke til at nul regnes med i de positive tal. Lad os tage et eksempel:

Eksempel 5: Binær repræsentation af negative tal

Tallet -11 lægges som BYTE som:

256 - 11 = 245 = $F5 = %11110101

Hov! tallet 245 kan altså forstås på to måder, alt efter om vi regner med fortegn eller ej. Vi har altså:

```
BYTE   $F5   = -11
UBYTE  $F5   = 245
```

Jeg ved dette kan virke forvirrende, men du er nødt til at optræne en vis rutine.

Som jeg var inde på i forrige kapitel, råder Amiga'en over en hvis mængde hukommelse, som på engelsk kaldes 'Memory'. Den del af hukommelsen vi er interesseret i hedder **RAM**. Amiga'en deler sin RAM op i to forskellige typer, nemlig **CHIP** eller Grafisk hukommelse, og **FAST** (=Hurtig) hukommelse. Grunden til disse to typer er, at Amiga'ens special-kredse kun kan finde ud af at bruge CHIP hukommelsen, mens CPU'en kan klare det hele.
Man kan altså godt have sit program liggende i FAST hukommelsen, men hvis du vil vise billeder eller spille musik, skal de data ligge i CHIP hukommelsen. Det er nemlig special-kredsene, der ordner alt, hvad der har med billede og lyd at gøre.

Som sagt skal alt, hvad du vil udføre, ligge i hukommelsen, men der er ikke noget i vejen for at smide noget ud, som du ikke længere skal bruge, og evt. hente noget nyt ind inden du benytter det. Dette kan blive nødvendigt, hvis man ikke har så meget hukommelse til sin rådighed, og hvad værre er; Amiga'erne kommer med forskellig mængde, alt efter hvilken model det drejer sig om. Nogle Amiga'er er født med 512 KB, andre med 1 eller 2 (eller flere) MB. 1 KB (Kilo Byte) indeholder 1024 bytes, altså ikke kun 1000 bytes. Dette skyldes jo computerens 2-talssystem, hvor 1024 er det nærmeste 'runde' tal til 1000, skrevet binært som %10000000000 = $400. Hvis man har 512 KB, har man altså

Fig 3: Er 1 kilo lig med 2 pund?
512*1024 bytes = 524.288 bytes.
1 MB (Mega Byte) er ligeledes 1024 KB, så
1 MB = 1024*1024 bytes = 1.048.576 bytes.

Vi tager lige et hurtigt overblik over Amiga'ens hukommelse:

Fig 4: Hukommelsesoversigt

$00000000	-	$0007FFFF	512 KB Chip RAM
$00080000	-	$001FFFFF	Reserveret til Chip RAM
$00200000	-	$009FFFFF	Mulig Fast RAM
$00A00000	-	$00BEFFFF	Reserveret
$00BFD000	-	$00BFDF00	PIA B (lige adresser)
$00BFE001	-	$00BFEF00	PIA A (ulige adresser)
$00C00000	-	$00DFEFFF	Reserveret for udvidelse (Fast)
$00DFF000	-	$00DFFFFF	Custom Chips (special kredsene)
$00E00000	-	$00E7FFFF	Reserveret
$00E80000	-	$00EFFFFF	Udvidelsesporte
$00F00000	-	$00F7FFFF	Reserveret
$00F80000	-	$00FFFFFF	System ROM

Jeg vil lige knytte et par enkelte kommentarer til denne liste. Har man mere end 512 KB Chip RAM, vil den resterende ligge fra $80000 og frem. F.eks. vil 1 MB Chip ligge fra $00000000 - $000FFFFF, og 2MB Chip vil befinde sig i hele området $00000000 - $001FFFFF. Dernæst er der problemet med RAM udvidelserne. Har du en A500 med 512KB udvidelse nede i bunden, vil denne hukommelse lægge sig fra $00C00000 - $00C7FFFF. Denne hukommelsesplacering går under navnet 'Slow RAM' (langsom RAM) grundet Amiga'ens systemopbygning. Sætter man ekstra hukommelse i en A2000, vil denne udvidelse placere sig fra $00200000 og frem, alt efter hvor meget hukommelse, man sætter i. Dette kaldes 'Ægte Fast RAM', da processoren er helt alene om at udnytte den.

Assemblersprogets format

I modsætning til en del høj-niveau sprog er Assembler linie-orienteret. Dette betyder, at der står præcis én komplet kommando på hver linie. Samtidig består hver linie af tre 'felter', nemlig et 'Label' felt, et instruktionsfelt og et 'Opcode' felt, på følgende måde:

Label Mnemonic Operand

Man benytter ikke linie-numre i assembler kode, så hvis man vil hoppe til en speciel linie (vi kommer tilbage til hvordan), giver man linien en 'Label' eller et navn. En labels første bogstav sættes helt op af den venstre margin, og nogle Assemblere vil gerne have at man sætter et kolon bagefter. Man behøver selvfølgelig ikke at give alle linierne en label; kun dem man er interesseret i at kunne finde senere.
Mnemonic's er det engelske navn for en assembler-kommando, og udtales Nu-monik. Her skrives selve kommando-navnet. Der skal enten være et kolon, et eller flere mellemrum eller en tabulering mellem en label og en mnemonic, ellers kan din Assembler ikke finde ud af, hvad det er, du har skrevet.
Efter endnu et (eller flere) mellemrum eller tabulering kommer så Operand'en. Alt efter hvor mange operander en kommando kræver, skrives de her, som regel adskilt med kommaer, men det kommer vi tilbage til, når vi rigtigt tager fat i programmeringen. Operanden indeholder groft sagt de tal og data, som kommandoen skal udføre.

Din CPU vil altså, når den skal udføre dit program, starte med den øverste linie, udføre kommandoen, springe videre til næste linie, udføre den osv. Her ses forskellen i overskuelighed mellem assembler og Maskinkode, for mens man i Assembler har en kommando pr. linie, ligger maskinkoden i én lang bit-streng i hukommelsen.

Man kan udvide linien med et fjerde felt, hvis man har lyst. Her kan man skrive en kommentar, til det der foregår i koden det pågældende sted, eller hvad hjertet nu kan begære. Man starter sin kommentar med et semi-kolon, der giver Assembleren besked om, at den ikke skal bryde sig om resten af linien. Hvis du derfor starter en linie med et semi-kolon, vil intet på den linie blive oversat til maskinkode. Nogle Assemblere tillader også, at man kan lave en hel linie kommentar ved at starte med en '*'. Pas derfor på ikke at starte et evt. label navn med dette tegn, da du så ikke får oversat linien. Kommentarer kan altså ikke ses i det oversatte program, og gør det derfor ikke længere.

Alt i alt ser en assembler-linie altså således ud:

Label Mnemonic Operand ;Kommentar

MC680x0 familien har et rigt udvalg af mnemonics (kommandoer) til data-flytning, aritmetik, logisk testning, bit manipulation og 'flow control', ialt ca. 56 instruktioner, men grundet 12 grundlæggende adresseringsformer giver det programmøren en utrolig masse frihed, og sammenlagt mere end 1000 muligheder. Med den mængde instruktioner er det samtidigt klart, at det samme program kan skrives på flere forskellige måder.

Til sidst skal du lige vide, at man angiver bredden af operationen sammen med sin mnemonic. Bredden kan være Byte, Word eller Long word, som angives med et .B, .W eller .L efter den pågældende mnemonic. Angiver man ikke nogen bredde, vil assembleren automatisk sætte bredden til Word, hvis kommandoen understøtter dette.

Et assembler-program, altså en liste af mnemonics, kaldes en 'Source code' eller en lille smule fordansket, en 'source-kode'.

Vores CPU har nogle indbyggede registre, som kan benyttes som midlertidige arbejdsområder. Det er ikke meningen man skal have data liggende i disse i længere tid ad gangen, og de tidligere beskrevne Opcodes består ofte af indeholdet af disse registre. Fidusen ved disse registre er netop, at de ligger internt i processoren, så der ikke skal bruges tid til at hente data fra RAM'en.

Der er forskellige typer af disse registre, bl.a. 8 data-registre, som hovedsagligt benyttes til at gemme aritmetiske værdier, tællere, index o.l. Data-registrene kaldes **D0, D1, D2, D3, D4, D5, D6** og **D7**, og kan indeholde alle bit-længder op til 32 bits, dvs. at man selv kan vælge om man vil benytte 1 bit, 8 bits (1 byte), 16 bits (1 word) eller 32 bits (1 long word). Man skal lægge mærke til, at hvis man kun lægger en byte ned i registeret, ændrer man kun de 8 mindst betydende bits, altså dem længst til højre, resten af registeret er uændret. Ligeledes vil et word lægge sig i de 16 mindst betydende bits, mens et long word vil fylde hele registeret.

Der findes tilsvarende 8 adresse-registre, der oftest benyttes som pointere til hukommelsen, men de kan også indeholde andre værdier. Navnene på disse registre burde ikke komme som noget chok for jer; **A0, A1, A2, A3, A4, A5, A6** og **A7**. Da man hovedsagligt benytter A-registrene til adressering, benytter disse kun længderne word og long word. Et long word vil selvfølgelig fylde hele registret ud. Det er lidt mere spidsfindigt, hvis du skriver en word-værdi ned i et A-register. Din værdi bliver lagt i det laveste word, ligesom ved D-registrene, men det øverste word bliver fyldt op med kopier af bit 15. Dermed ødelægges hele registeret i alle tilfælde. Kopien af bit 15 gør, at dit word gemmes med fortegn hvad enten du vil eller ej; er bit 15 sat (=1) vil processoren behandle det som et negativt tal og fylde bit 16-31 op med 1-taller. Denne behandling af words gør, at A-registre også kan benyttes til index-værdier.

Udover de to nævnte typer registre, findes der nogle enkelte specielle af slagsen. Først og fremmest er der et register, der hedder **PC** - Program Counter. Dette er et internt 32-bit register, som CPU'en bruger til at pege på det sted i hukommelsen, den er i gang med at udføre en kommando. Når du henter et program ind i hukommelsen, sætter CPU'en PC til at pege på starten af dit program, og begynder at udføre kommandoerne derfra mens den hele tiden rykker PC videre i dit program. Vi piller ikke direkte ved dette register, men lader pænt CPU'en afgøre, hvad der skal stå i det.

Processoren har undertiden behov for noget ekstra albuerum til at gemme vigtige oplysninger. Dette gør den i et område af RAM'en, som den benytter som '**Stack**'. Da denne stack kan ligge hvorsomhelst (hvert program får automatisk tildelt en stack når det startes op), har CPU'en et register til at pege på stacken. Dette register er faktisk A7 registret, så det er bedst at holde fingrene fra dette. A7 kaldes også **SP**, som står for Stack Pointer. Faktisk har processoren to stacks, men den sørger selv for at holde rede i dem, og benytter A7 som pointer for dem begge, idet kun én af disse stacks kan benyttes ad gangen, alt efter hvilken 'mode' processoren arbejder i. Vi vil ikke komme nærmere ind på dette i denne

bog, så lad os bare aftale, at A7 er Stack Pointeren.

Endelig er der nogle registre, der ændrer sig, alt efter hvad computeren foretager sig. Det vigtigste for os er **CCR**, som står for Condition Code Register. Dette register indeholder en række flag, som sættes eller slettes alt efter hvilket resultat, den sidst udførte kommando gav. Hvis man f.eks. fik et negativt resultat af en beregning, ville N (Negative) flaget være sat. Dette er meget brugbart, og du kommer til at se det anvendt senere i bogen. CCR ser således ud:

Fig 5: Oversigt over CCR

Bit	7	6	5	4	3	2	1	0
Symbol	-	-	-	X	N	Z	V	C

Hvor flagene er sat, hvis en operation har:

C = Carry	- Givet en mente
V = oVerflow	- Givet et for stort tal med fortegn
Z = Zero	- Givet resultatet nul
N = Negative	- Givet et negativt resultat
X = eXtended	- Givet mente ved aritmetiske operationer
- =	- Flaget er ubenyttet

Selvom jeg ikke vil komme ind på betydningen af **SR** registret i denne bog, vil jeg for en ordens skyld lige nævne indholdet. **SR** er processorens Status Register, og CPU'en benytter sig af dette til at markere, hvilken status den er i for tiden. Dette register kan du kun komme i kontakt med, hvis du har sat CPU'en til at køre i noget, der kaldes '**Supervisor Mode**'. Normalt kører vi i det, der kaldes '**User Mode**' (bruger funktionsmåde). Her er indholdet af registret, som i praksis ligger som den mest betydende byte i CCR (derfor de høje bit-numre):

Fig 6: Oversigt over SR

Bit	15	14	13	12	11	10	9	8
Symbol	T	-	S	-	-	I2	I1	I0

Hvor:

T = Trace mode - CPU'en kører i Trace mode
S = Supervisor - CPU'en kører i Supervisor mode
I = Interrupt - Angiver CPU'ens mulige levels

Da hver kommando har mange muligheder for både adressering gennem operanderne og databredder, vil jeg i denne bog benytte en fast syntax for hver kommando. Først og fremmest er der databredden; den bliver opgivet hver gang, men skrives således:

Mnemonic.s <operand(er)>

hvor 'mnemonic' er den aktuelle instruktion og '.s' angiver bredden, altså enten **.B** (byte), **.W** (word) eller **.L** (long word).
Derudover er der operanderne. Du vil se dem præsenteret på forskellige måder:

s ;Source, kilde ('fra')
d ;Destination ('til')

<ea> ;Effective Address
Dn ;Et vilkårligt D-register
An ;Et vilkårligt A-register
Rx ;Et vilkårligt register
Ry ;Et vilkårligt register, men ikke nødvendigvis det samme som Rx

Når du ser disse betegnelser, er det meningen, at du skal proppe den operand du har brug for, ind på pladsen. Her er der mange muligheder:

01 Dn	**- Vilkårligt data-register**
02 An	**- Vilkårligt adresse-register**
03 (An)	**- Indholdet af adresse**
04 (An)+	**- Indholdet af adresse med increment (forøgelse)**
05 -(An)	**- Indholdet af adresse med decrement (formindskelse)**
06 d16(An)	**- Indholdet af adresse med 16 bit offset (forskydning)**
07 #<data>	**- Konstant**
08 d16(PC)	**- Indholdet af nuværende adresse med 16 bit offset**
09 (An,Rx.s)	**- Indholdet af adresse med offset indeholdt af Rx**
10 d8(An,Rx.s)	**- Indholdet af adresse med 8-bit offset samt offset indeholdt i Rx**
11 xxxx.L	**- Indholdet af fast adresse**
12 xxxx.L	**- Fast adresse**

Læg mærke til at vi ikke skelner syntax-mæssigt i de to sidstnævnte tilfælde. Dette skyldes, at kommandoerne har deres egen opfattelse af hvad du mener, men det kommer vi altsammen tilbage til. Når jeg senere i bogen forklarer om de forskellige kommandoers adresseringsmuligheder, vil jeg kun referere til de numre, der står forrest på linien. Jeg ved dette er lidt besværligt for læseren i starten, men det er desværre nødvendigt af pladshensyn i denne bog.
Jeg ved godt, at dette ser temmelig kryptisk ud på nuværende tidspunkt, men der vil blive kastet lys over operanderne i de næste par afsnit.

Assembler direktiver

Vi skal lige se på nogle kommandoer, som egentlig ikke findes i maskinkode, men som vores Assemblere er udstyret med for at lette vores arbejde. Disse direktiver benyttes blandt andet til at definere konstanter, variable og fast lagerplads i programmet.

En konstant kan defineres en gang for alle i starten af et program, så man kun skal rette den i definitionen for at hele programmet følger med:

```
Bredde EQU 320
Hoejde EQU 256
Dybde  EQU   4
```

EQU betyder 'EQUivalent', hvilket er det engelske ord for 'ensbetydende' eller 'samme værdi'. Værdierne i dette eksempel kunne være nogle konstanter til et program, der har skærmens størrelse indkodet, her en lo-res skærm (320 x 256 pixels i 4 bitplaner). Efter programmet er lavet, finder man ud af, at man hellere vil lave skærmen i med-res, så man ændrer bare den øverste linie til:

```
Bredde EQU 640
```

og assemblerer igen; programmet sætter nu en med-res skærm op i stedet for. Læg mærke til at man bliver nødt til at assemblere programmet igen, for at ændringerne træder i kraft - man kan ikke skifte dem inde fra sit program; det er jo konstanter. Hver gang Assembleren møder konstantens navn, f.eks. 'Dybde', sætter den simpelthen værdien 4 ind i stedet for. Skal man f.eks. bruge denne skærms bredde i byte i stedet for pixels, kan man benytte lidt matematik i linien:

```
Bytes  EQU Bredde/8
```

Vil man hellere have en variabel, må man lægge den i RAM'en. Dette kan gøres med:

```
Bredde: dc.w   320
```

'**DC**' står for 'Define Constant data' (definer konstant data), hvilket egentligt er noget vrøvl, når det drejer sig om en variabel. Ikke desto mindre er 'Bredde' nu en variabel, der er initialiseret til værdien 320, f.eks. i en IFF-billed-fremviser. Efter at have læst billed-data finder programmet ud af, at billedet er i med-res, og derfor skriver den bare værdien 640 ned i 'Bredde', som nu har denne værdi øjeblikkeligt. Programmet er til fulde herre over indholdet i variable, og de kan ændres, så meget man vil. Bid iøvrigt mærke i, at man skal sætte bredden af variablen, når man definerer den; her er bredden et word (.w), og det kan programmet ikke udvide. Kender man derfor ikke det største tal, som skal ned i variablen, må man hellere definere et long word, og så kun bruge den plads man har behov for.

Man kan definere flere variable i samme linie, f.eks. til en tabel:

```
Tabel: dc.b 1,2,3,4,5,6,7,8,9,10
```

som i dette tilfælde vil lave en tabel med de nævnte værdier. Hver værdi fylder en byte, så hele tabellen fylder altså 10 bytes.
Man kan også gemme en tekst på denne måde:

```
Tekst: dc.b 'Denne tekst skal gemmes',0
```

Assembleren vil oversætte alle bogstaverne til nogle tal-værdier (bytes) og gemme dem. Disse tal-værdier er nøje defineret i ASCII-standarden, så der burde ikke komme overraskelser. Ydermere bliver der lagt et 0 som afslutning af teksten, men dette er selvfølgelig valgfrit, det var mest taget med for at vise, at tekst og tal kan defineres i samme linie.
Hvis vi skal kende længden af denne tekst-streng, kan vi jo vælge at tælle karaktererne, men hvis teksten ændres på et senere tidspunkt, bliver der hurtigt en frygtelig masse ændringer i din source-kode. Derfor kan man lave en konstant, som altid bliver opdateret, når programmet oversættes:

```
Tekst: dc.b 'Her er 20 karakterer'
Antal  EQU  *-Tekst
```

Konstanten 'Antal' bliver hermed sat til antallet af karakterer i teksten. Dette gør Assembleren, fordi '*' betyder, at den skal tage den nuværende adresse (PC) og trække start-adressen for 'Tekst' fra. Dermed opnår den forskellen i bytes mellem de to adresser, og det er lige netop længden af teksten!

Vil man lave et større område med ens data, kan man slippe lidt lettere over det ved at benytte direktivet '**DCB**', som står for 'Define Constant Block' (definer konstant-blok), altså en blok af tal, eller en tabel. Igen siger navnet 'konstant' og igen drejer det sig faktisk om en variabel. Enkelte gamle assemblere benytter direktivet '**BLK**' i stedet for, og man kan vel gætte på at det betyder 'BLocK'; i hvert fald er syntaxen den samme som ved 'dcb':

```
Lager: dcb.l 120,$30
```

Her laves et område, 'Lager', bestående af 130 long words, alle med værdien $30.

Har man bare behov for lidt arbejdsplads eller et data-område, man frit kan benytte til forskellige værdier, men er ligeglad med, hvad der ligger i det fra starten, kan dette gøres med:

```
FriPlads: ds.b 10000
```

På adressen 'FriPlads' indsætter Assembleren et område, der er 10.000 bytes langt. Pas på med at tro at der bare ligger nuller i dette område, for i princippet

kan der ligge hvadsomhelst af affald fra et tidligere program, der lå der. Skal du bruge nuller, er det derfor sikrest at slette hele området manuelt, men det kommer vi ind på senere. '**DS**' står for 'Define Storage' (definer lagerplads).

Inden du laver lange data-områder med en skønsom blanding af bytes, words og long words, skal du lige huske på, at kun bytes kan lægges på en ulige adresse. Din Assembler vil gøre vrøvl og komme med en surmulende meddelelse om, at du prøver at lægge et word på en ulige adresse, hvis du ikke selv er opmærksom på det. For at afhjælpe dette, har Assembleren et par direktiver:

EVEN

Sørger for at næste kommando står på en lige adresse, f.eks:

Eksempel 6

```
        dc.w    0       ;Lige adr.
        dc.l    0       ;Lige adr. +2
        dc.b    0       ;Lige adr. +6
        EVEN            ;Ulige adr. +7
        dc.w    0       ;Lige adr. +8
```

I dette eksempel har Assembleren sat et nul ind på EVEN's plads og dermed rykket det sidste word ned på en lige position, så alle er glade (Even betyder 'lige'). Med andre ord sørger EVEN for, at addresse-nummeret kan deles med 2. Dette kaldes på engelsk for 'Word Align'. I sjældne tilfælde (især i forbindelse med system-programmering) skal adressen være det man kalder 'Long aligned', altså adresse-nummeret skal være deleligt med 4. Dette sker ved en kommando, de fleste Assemblere kalder:

CNOP x,y

som står for 'Conditional NOP' (NOP er en form for 'tom' instruktion som kaldes 'No OPeration' og ikke udfører nogetsomhelst). Instruktionen sørger for at næste adresse bliver delelig med y, og lægger derefter x til adresse. Skal man lave et 'long align' gøres det altså med:

CNOP 0,4

Samtidig kan du se, at man kan benytte kommandoen

CNOP 0,2

istedet for EVEN.
Flere direktiver kommer vi ikke ind på i denne bog.

Hører jeg dig grine derude? Ja, der såmænd god grund til det, men jeg tør ikke lære dig at lave programmer, før jeg har forvisset mig om, at du også kan komme ud af dem igen, uden at tilkalde Guru'en. Ergo; vi starter med slutningen.

Du skal såmænd kun lære én instruktion i dette kapitel, men som før nævnt er den essentiel for at kunne afslutte et program. Glemmer du at give processoren besked om, at programmet er slut, vil den bare fortsætte med at fortolke instruktioner, der ikke er der. Dette vil gøre den så forvirret, at den går totalt i koma, og på et eller andet tidspunkt vil den bede Guru'en om hjælp. Og lad os da lige blive enige om, at det virker temmeligt uprofessionelt at afslutte sit program med en blinkende, rød kasse.

Så husk nu at afslutte dit program med denne instruktion:

```
RTS     ;ReTurn from Subroutine
```

Det er tre sølle bogstaver, der kan betyde knald eller fald! Som du kan se betyder 'RTS' 'ReTurn from Subroutine', men normalt kalder vi bare kommandoen for 'Return', som betyder 'returnere' eller 'vende tilbage'. Det lange navn antyder dog, at det ikke kun er CLI'en, den kan returnere til, men det skal vi først se på meget senere i denne bog.

Nu må du hellere holde øjnene åbne, for vi skal til at møde den vigtigste kommando blandt Motorolas mnemonics, nemlig **MOVE**, som betyder 'flyt'. Du har nok gættet, at det er denne kommando, vi skal have fat i hver gang, vi skal rykke data rundt i hukommelsen eller i registrene. Det sjove ved valget af navn til denne kommando er, at 'flyt' skal læses med betydningen 'kopier', idet den ikke sletter tallet, hvor det oprindeligt stod, men bare laver en kopi af det i destinationen. MOVE findes i forskellige afskygninger, så lad os glane lidt på syntaxen:

MOVE.s s<ea>,d<ea>

hvor .s er .B, .W eller .L, s<ea> kan være samtlige operander fra listen, hvorimod d<ea> kun kan være 1,3,4,5,6,9,10,11,12.
Hvis vi f.eks. skal lægge tallet 23 over i både D0 og D1, kan vi gøre det på flere forskellige måder:

```
MOVE   #23,D0
MOVE   #23,D1
```

eller

```
MOVE   #23,D0
MOVE   D0,D1
```

Den nederste metode er, om ikke den smarteste, så i hvert fald den hurtigste. Læg mærke til, at der er et '#' tegn foran tallet 23. Det angiver, at det kommende tal er en konstant (Immediate data). Du kan selvfølgelig også angive tallet i hex- eller binær notation, så husker du bare '$' eller '%' tegnet også, f.eks:

```
MOVE   #$17,D0
MOVE   #%10111,D1
```

Ligeledes kan du benytte en konstant:

```
MOVE   #Bredde,D0
```

Hvis du glemmer '#' tegnet, sker der noget temmelig anderledes, nemlig at indholdet af adresse 23 bliver lagt i registret i stedet for konstanten 23. Indeholdet af adresse 23 kan være temmelig tilfældigt, så hvad du får ud af kommandoen:

```
MOVE.B  23,D0   ;Kun byte, ulige adr.
```

er et godt spørgsmål. Normalt skriver vi adresser som Hex long words, altså i dette tilfælde:

```
MOVE.B  $00000017,D0   ;$17 = 23
```

På den måde er det nemmere at se, at det er en adresse, selvom vi kun flytter en

byte over i D0 (man kan jo kun flytte words og long words fra lige adresser, og det er $17 jo ikke). Prøver du at lave kommandoen:

```
MOVE   #$00000017,D0
```

skulle din assembler meget gerne gøre vrøvl, når du prøver at oversætte programmet, fordi der er angivet et long word i konstanten, men kun et word i bredden. Lad os lige genopfriske, hvordan MOVE ter sig mht. de forskellige databredder:

```
1 MOVE.B  #$17,D0   ;D0=$xxxxxx17
2 MOVE.W  #$17,D0   ;D0=$xxxx0017
3 MOVE.L  #$17,D0   ;D0=$00000017
```

hvor et 'x' betyder, at cifret ikke bliver påvirket af operationen. Indeholdt D0 f.eks. tallet $AAAAAAAA, ville tallet i D0 være følgende efter hver af de tre linier:

1 D0 = $AAAAAA17
2 D0 = $AAAA0017
3 D0 = $00000017

... og så gentager vi heller ikke dette mere!

Vil du kopiere en konstant over i et data-register, kan man for små konstanter benytte:

MOVEQ #<data>,Dn ;MOVE Quick

Begrænsningen er, at <data> skal være en signed byte, altså ligge mellem -128 og 127, men tallet vil blive lagret som et long word. F.eks. vil følgende:

```
MOVE.L #$AAAAAAAA,D0
MOVEQ  #-1,D0
```

resultere i at D0 nu indeholder $FFFFFFFF (-1), og altså ikke $AAAAAAFF. MOVEQ er hurtigere at udføre for processoren, og det er så fordelen, f.eks. er den hurtigste metode at slette et D-register v.h.a.:

```
MOVEQ  #0,Dn
```

Skal man have fat i en adresse, er den rigtige syntax

MOVEA.s <ea>,An ;MOVE Address

hvor <ea> dækker hele listen, mens .s kun kan være .W og .L

Begrænsningen ligger, som du kan se, i at kommandoen kun kan benyttes på A-registre; det er jo ikke uden grund, at de netop hedder Adresse-registre!

Der er flere MOVE kommandoer, men vi skal kun se på én til, nemlig:

MOVEM.s #<data>,<ea>
MOVEM.s <ea>,#<data>

hvor .s er enten .W eller .L
MOVEM står for MOVE Multiple, altså en kopiering af mere en én operand til en destination. Dette gøres ved at skrive en hel liste på <ea>'s plads, og det benyttes hovedsagligt til at smække registre på stacken, f.eks:

MOVEM.L D0/D1/D2/D3/A0/A6,-(SP)

der gemmer alle de nævnte registre, D0, D1, D2, D3, A0 og A6, på stacken. Man kan skrive den samme kommando lidt lettere:

MOVEM.L D0-D3/A0/A6,-(SP)

Man kan benytte dette, hvis man har vigtige data i sine registre, men lige skal udføre en rutine, der netop ødelægger disse registre. Efter rutinen er udført, kan man få alle sine data tilbage ved at skrive:

MOVEM.L (SP)+,D0-D3/A0/A6

og vupti har man sine værdier på plads i registrene.

Vi skal også lige se på en meget speciel udnyttelse af MOVE, nemlig til at sætte flagene i CCR (og SR) registret. Man bruger dette meget sjældent, men for en god ordens skyld, vil jeg nævne at man kan:

MOVE <ea>,CCR

hvor <ea> kan være alt på nær A-registre, hvilket vil sige 1,3,4,5,6,7,8,9,10,11,12 i listen i afsnittet om syntaxen. Lad os tage et par eksempler:

MOVE #$1F,CCR ;Tænd alle flag
MOVE #%00100,CCR ;Tænd Z (Zero) flaget, bit 2

På samme måde kan SR påvirkes, men det kræver kendskaber, der ligger uden for denne bogs rammer.

Vi skal se meget mere til MOVE kommandoerne senere i denne bog, men først skal vi have lidt bedre kendskab til instruktionssættet.

Specielle kommandoer

Vi skal også se på et par kommandoer, der bærer præg af dataflytning, men er af lidt mere speciel karakter. Den mest simple af disse kommandoer, er

CLR.s <ea> ;CLeaR an operand

hvor .s dækker de tre muligheder .B, .W og .L og <ea> tager det meste af listen, nemlig 1,3,4,5,6,9,10,11,12. Kommandoen sletter simpelthen operanden ('clear' betyder 'slet'), og fungerer dermed på samme måde som kommandoen 'MOVE.s #0,<ea>'.
Normalt er MOVE instruktionen at foretrække, idet CLR både udfører en læse- og en skrive operation. Dette gør den en smule langsommere at udføre, men det er totalt uskadeligt at benytte den, og en gang imellem ligefrem at foretrække.

Lad os springe hurtigt videre til den næste kommando:

EXG Rx,Ry ;EXchanGe registers

som bytter om på indholdet af to registre. Sjovt nok betyder 'exchange' faktisk 'ombytte', så det skulle ikke være årsag til nogen større forvirring! Kommandoen opererer altid på long words, så man angiver ikke nogen data-bredde. Lad os tage et eksempel:

```
movea.l #$20000,A0 ;Læg værdi i A0
moveq  #0,D1       ;Slet D1
exg    A0,D1       ;Byt om på A0 og D1
rts                ;Afslut program
```

Efter kørsel af programmet vil A0 være 0 og D1 være $20000.

Lad os tage en sidste kommando i dette afsnit, nemlig

SWAP Dn ;SWAP register halves

Denne instruktion bytter om på halvdelene i et D-register. 'Swap' betyder 'bytte', men man må ikke forveksle denne kommando med den førnævnte EXG. Som syntaxen antyder, bytter EXG om på to operander, mens SWAP bytter om inden i én operand. Denne ombytning betyder, at lo-delen af dit long word bliver lagt op i high-delen og omvendt. Det kræver vist et eksempel:

```
move.l #$11119999,D0 ;Læg værdi i D0
swap   D0            ;Ombyt halvdelene i D0
rts                  ;Afslut
```

Når dette program er udført, vil D0 indeholde tallet $99991111. Det lyder måske som en total åndsvag kommando, men den er nu slet ikke så tosset. Vi kommer til at se den anvendt, når vi begynder at dividere tal med hinanden.

Adressering

Tidligere i bogen blev RAM'en sammenlignet med en lang vej med en entydig adressering, præcis som i det virkelige liv. Forskellen er, at når man vil i kontakt med en adresse i det pulserende liv, sender man bare et brev - prop et frimærke på, og postvæsenet skal nok finde frem til vedkommende. Så let er det desværre ikke i assembler. Her er vi nødt til selv at banke på døren, og dette kapitel kommer ind på hvordan vi finder adressen.

Man benytter hovedsagligt et A-register til at pege på den adresse, vi vil have fat i, men så snart den er fundet, har vi rigelige muligheder for at specificere, hvad vi vil gøre med den adresse. Vi har allerede set på én metode til at fange en adresse, nemlig v.h.a. MOVE og MOVEA kommandoerne, men der findes en anden og bedre:

```
LEA <ea>,An   ;Load Effective Address
```

hvor <ea> er enten 3,6,7,8,9,10,11 eller 12. Da det er en adresse denne kommando henter til A-registret, påvirkes dette selvfølgelig kun som long word, så der skal ikke angives en bredde - den er underforstået. Et kort eksempel:

Eksempel 7A

```
01        lea       Tabel,A0       ;Peg på tabellen
02        move.b    (A0)+,D0       ;Hent første tal
03        move.b    (A0)+,D1       ;Hent andet tal
04        move.l    (A0),D2        ;Hent et long word
05        move.b    -(A0),D3       ;Hent andet tal
06        rts                      ;Afslut
07
08 Tabel: dc.b      0,1,2,3,4,5
```

Herefter vil D0 indeholde tallet $00, D1 vil være $01, D2 er $02030405 og D3 skulle gerne være $01, idet vi benytter indirekte adressering. Linie 01 henter den konkrete adresse på tabellen (der peges altså på byte nummer 0 i tabellen), linie 02 henter den byte, der står på denne adresse, og rykker A0 til at pege på den næste byte. Linie 03 henter derfor byte nummer 1, som er $01, rykker igen adressen i A0 til byte nummer 2. Herefter snupper vi det long word, der starter her, altså $02030405, i linie 04, men denne gang øger vi ikke adressen, som A0 peger på. Nu sker den omvendte process, idet linie 05 først trækker én byte fra adressen i A0, og derefter henter værdien. Da A0 peger på 'Tabel+2' efter linie 04, vil vi altså først lade A0 pege på 'Tabel+1' og derefter hente værdien, altså $01.

Vi ser altså, at et sæt paranteser omkring et A-register betyder, at man i stedet for selve værdien af registeret, tager indholdet af den adresse, det peger på. De forvirende '+' og '-' tegn efter og før operanden, betyder at man ændrer A-registeret til at pege på den følgende/foregående adresse. Man skal læse det stringent i læse-retningen: Minus kommer før operanden, altså ændres adressen før man tager indholdet, plus kommer efter operanden og dermed tages indholdet først. Ændringen af adressen er afhængig af den længde, du angiver. Er bredden

en byte, vil adressen blive ændret med en byte, er bredden et word, vil adresse blive ændret et word... Gæt selv hvad der sker med long words.

Det samme program kunne skrives med absolutte forskydninger sammen med den indirekte adressering:

Eksempel 7B

```
01        lea       Tabel,A0          ;Peg på tabellen
02        move.b    (A0),D0           ;Hent første tal
03        move.b    1(A0),D1          ;Hent andet tal
04        move.l    2(A0),D2          ;Hent long word
05        move.b    1(A0),D3          ;Hent andet tal
06        rts                         ;Afslut
07
08 Tabel: dc.b      0,1,2,3,4,5
```

Dette program vil udføre præcis det samme, som det ovenstående, men vi ændrer ikke på den adresse, der står i A0. Efter program nummer 2 er kørt, vil A0 stadig pege på 'Tabel', hvor A0 i program nummer 1 vil pege på 'Tabel+1', når programmet er kørt. Forskellen er, at man i stedet for at tælle A0 op, så den hele tiden peger på det tal, man skal bruge, angiver hvor langt man skal gå ud fra denne adresse i stedet for. Linie 03 siger derfor, at man skal tage den byte, der står én byte længere henne end adressen i A0. Tilsvarende i linie 04 angiver man en forskydning på 2 bytes. Man kan altså vælge den teknik, man har behov for, alt efter om man skal bruge adressen på tabellen senere, eller om man er ligeglad med den, når man er færdig med at læse sine tal.

En forskydning angives altid i bytes, ligemeget hvad data bredden er. Derfor skal du huske, at words og long words skal ramme en lige adresse. Hvis A-registeret peger på en ulige adresse, skal forskydningen altså også være et ulige antal bytes og omvendt.

Hvis tabellen ligger inden for overskuelig afstand, 32KB, kan man adressere PC afhængigt. Man ændrer bare linie 01 til:

```
01    lea    Tabel(PC),A0 ;Peg på tabellen
```

og fortsætter herefter uforknyt. Dette er lidt hurtigere at udføre for processoren, men som sagt kan man så ikke nå overalt i hukommelsen, men kun et Signed Word (32KB) fra den adresse man står på, som jo er indeholdt i PC. Adressering ud fra PC kan kun ske i source-operanden, aldrig i destination-operanden.

Adressering

Der er et par avancerede adresseringsmuligheder tilbage, der på det nærmeste blander hele dynen af muligheder sammen i en stor pærevælling. Men det skal jo ikke afskrække os fra at kigge dem nærmere i sømmene. Hvis A0 stadig peger på tabellen, og D0 indeholder den position i tabellen, man gerne vil aflæse, kan dette gøres med:

```
move.b  (A0,D0),D1
```

I dette tilfælde indeholder A0 adressen, hvor tabellen starter, og D0 indeholder forskydningen. Hvis vi ændrer linie 03 i ovenstående program til

```
03      move.b (A0,D0),D1
```

ville D1 indeholde tallet 0. Det skyldes, at D0 i linie 02 får indlæst det første tal i tabellen, der jo er nul. Dermed giver det 0 bytes forskydning af A0, og D1 får også det første tal i tabellen. Derfor må vi gøre et eller andet for at få programmet til at virke med den nye metode. Dette 'et eller andet' kunne jo passende være at genindføre vores direkte forskydning igen:

```
03      move.b 1(A0,D0),D1
```

Og så er vi tilbage hvor vi kom fra. I dette tilfælde er det godt nok lidt af en omvej, men det er smart i andre tilfælde.
Når vi skriver (A0,D0) vil indholdet af D0 som sagt blive brugt som indirekte forskydning, men i dette tilfælde vil D0 blive tolket som et UWORD. Ønsker man at tage hensyn til et evt. fortegn, tilføjer man lige et .W, (A0,D0.W).

Jeg bør lige nævne, at man også kan benytte MOVE, til finde en pointer til en adresse. Tidligere så vi, at hvis man kender adressen, kan man bare hente en tilsvarende værdi til registeret, f.eks 'MOVEA #$50000,A0', der lægger adressen $00050000 i A0. Problemet er, at vi sjældent kender den faste adresse, men også det er der råd for. Her skal vi finde vores gode '#' tegn frem igen, hvorefter linie 01 kan erstattes med:

```
01      movea.l #Tabel,A0
```

Det er dog kun, når man skal have en adresse over i et D-register, at denne metode er at foretrække (man skal så selvfølgelig benytte 'move.l' i stedet for).

Det var en ordentlig mundfuld, men fortvivl ikke. Du kommer nok til at springe tilbage til dette kapitel et par gange, for at få styr på adresseringen efterhånden som du kommer gennem bogen. Men som man siger: Øvelse gør mester!

Normalt udføres et program, instruktion for instruktion, i en sekvens. MC680x0 henter den instruktion, som PC (Program Counter) peger på, ændrer PC til at pege på næste instruktion, og så fremdeles. Det er også godt nok, men nogle gange bliver man nødt til at ændre den sekventielle udførsel, og hoppe hen til et andet sted i programmet. Der findes to forskellige typer af forgreninger, nemlig de ubetingede og de betingede. De ubetingede forgreninger stiller ikke nogen krav til nogetsomhelst; de hopper bare derudaf, når du giver besked på det, mens de betingede forgreninger kun hopper, hvis en eller flere betingelser er opfyldt.
Lad os først se på den ubetingede forgrening. Her findes to kommandoer:

JMP <ea> ; JuMP (hop)
BRA.s <label> ; BRanch Always (forgren)

Den altafgørende forskel mellem de to kommandoer er, at JMP udregner en adresse at hoppe til (<ea>), og lægger den over i PC. Dermed kan JMP kommandoen hoppe rundt i hele din RAM, hvorimod BRA er PC-relativ, det vil sige at den lægger <label> til PC. Da værdien af <label> maximalt er beskrevet af et word, kan man altså højest hoppe over afstande på 32KB (32728 bytes). Normalt benyttes .s ikke og giver dermed BRA den føromtalte rækkevidde, men ønsker man kun at hoppe over små afstande, der kan beskrives med en byte, altså 128 bytes, kan .s være enten .B eller .S som er to alen af ét stykke. Personligt foretrækker jeg .S versionen, som står for Short (kort), .B hentyder til bredden af det tal, der lægges til PC, altså en byte.
Vi kalder en lang forgrening (mere end 32KB) for et 'hop', og ellers kalder vi det en forgrening. Vi må hellere tage et par korte eksempler:

```
JMP    $50000    ;Hop til adresse $50000
JMP    InitDemo  ;Hop til label InitDemo
BRA    StartDemo ;Forgren til label StartDemo
BRA.S  Verify    ;Forgren kort til label Verify
```

Betingede forgreninger

Det er ikke altid smart at udføre et hop eller en forgrening ubetinget. Ofte vil man kun forgrene, hvis en betingelse er opfyldt, og tro mig, MC680x0 har en lang række muligheder at byde på. Faktisk har Motorola været ovenud gavmilde på dette punkt!
Før fortalte jeg om BRA kommandoen, der altid forgrener, men der er gudskelov mere end én branch kommando. Faktisk en hel liste, der dækker stort set alle kombinationer af Condition flagene.

Forgreninger og hop

Idet vi snakker om branch (forgrening) kommandoer, der er betingede af vores Condition Codes, samler vi dem under navnet **Bcc**, som står for Branch if condition code (endelig ikke at forveksle med BCC). Her kommer mulighederne:

Fig8: Oversigt over condition codes

cc	Betydning	cc	Betydning
EQ	EQual	NE	Not Equal
GT	Greater Than	LT	Less Than
GE	Greater or Equal	LE	Less or Equal
CC	Carry Clear	CS	Carry Set
HI	HIgher	LO	LOwer
HS	Higher or Same	LS	Lower or Same
PL	PLus	MI	MInus
VC	oVerflow Clear	VS	oVerflow Set

(Husk at mnemonic'en har et B foran betingelsen, f.eks. BEQ)

Som du kan se, er der nævnt 16 forskellige betingelser, vi kan stille, men der ligger lidt skjult svindel i listen. BHS er nemlig synonym for BCC ligesom BLO er det for BCS, så reelt giver det os 14 forskellige muligheder. Fidusen er, at alt efter hvad der sker i assembler koden det pågældende sted, kan den ene skrivemåde være mere beskrivende end den anden - de fleste Assemblere skal nok oversætte det rigtigt.

Prøv at køre følgende lille program, med forskellige værdier af <tal>:

Eksempel 8

```
           move.l    <tal>,D0
           bmi.s     Negativ
           move.w    D0,D1
           bmi.s     Negativ2
           rts
Negativ:   moveq     #-1,D1
           rts
Negativ2:  move.w    #-1,D1
           rts
```

Læg mærke til hvad der returneres i D1. Forsøg dig med værdier som $1f000 og $ff000000.

Langt de fleste instruktioner påvirker CCR-registeret, så her har vi åbenbare muligheder for at teste, hvordan vores operationer faldt ud. Vi tager lige en simpel ny kommando som eksempel:

```
TST.s <ea> ; TeST an operand
```

Denne kommando kan udføres for '.s' som både bytes, words og long words, og <ea> kan både være et register eller en adresse i hukommelsen. Funktionen er simpel; den undersøger om <ea> er nul, og hvis den er, sættes Z (zero=nul) flaget true (Z=1), ellers slettes det, f.eks kan man finde det første nul i et dataområde:

Eksempel 9

```
1        lea     Tabel,A0                    ;Peg på område
2 Loop:  tst .w  (A0)+                       ;Er tallet 0?
3        bne.s   Loop                        ;Nej, prøv igen
4 ;Vi har fundet et 0 (juhu)!
5        rts
6
7 Tabel: dc.w    $1234,$4321,$ABCD,$0000,$9999
```

Når dette lille program slutter, vil A0 pege på tallet $9999 i tabellen, da A0 allerede bliver opdateret til at pege videre i linie 2.

Der findes gudskelov andre metoder til at sammenligne data, nemlig ved hjælp af sammenligningskommandoen, Compare (sammenlign), der findes i flere afskygninger:

```
CMP.s <ea>,Dn        ;CoMPare
CMPA.s <ea>,An       ;CoMPare Address
CMPI.s #data,<ea>    ;CoMPare Immediate
CMPM.s (An)+,(An)+   ;CoMPare Memory
```

hvor .s er enten .B, .W eller .L
I de fleste Assemblere er det godt nok bare at skrive CMP, så indsætter assembleren selv den rigtige version, når den oversætter, men til tider er det fornuftigt at skrive den rigtige version selv. På den måde kan man hurtigere se, hvad det er, man er interesseret i, hvis man kigger sin kode igennem på et senere tidspunkt. Man skal også passe lidt på CMPM kommandoen, som bliver sat automatisk i DevPac assembleren, men ikke i AsmOne.
Compare sammenligner altså to tal, og husker resultatet af denne sammenligning ved hjælp af sine Condition Codes, så nu er det bare op til os at benytte den forgreningstype, vi er interesseret i.

Sammenligninger

Lad os lave en rutine, der sammenligner to data-områder, som er lige store:

Eksempel 10

```
Verify:   lea      Data1,A0       ;Peg på område 1
          lea      Data2,A1       ;Peg på område 2
          lea      DataEnd,A2     ;Peg på slut af data
V_Loop:   cmpm.l   (A0)+,(A1)+    ;Sammenlign et longword
          bne.s    Error          ;Hop hvis ikke ens
          cmp.l    A1,A2          ;Er vi færdige
          bne.s    V_Loop         ;Nej, næste long word
          moveq    #0,D0          ;Alt gik godt
          rts                     ;Afslut rutine

Error:    moveq    #20,D0         ;Sæt fejl-meddelelse
          rts                     ;Afslut rutine

Data1:    ds.l     300            ;Område 1
Data2:    ds.l     300            ;Område 2
DataEnd:
```

Som du kan se, har vi brugt de betingede forgreninger til at lave en løkke med i begge eksempler. Vi skal senere se, hvordan man kan lave løkker på en anden måde.

I assembler har vi kun de fire regningsarter, plus, minus, gange og dividere direkte til rådighed. Skal man lave andre beregninger, f.eks. kvadratrødder eller trigonometriske funktioner, må vi konstruere beregningerne selv. Dette ligger dog udenfor denne bog.

Adition

Der er en del forskellige kommandoer til addition af tal. I denne bog vil vi se på:

```
ADD   ;ADD binary
ADDI  ;ADD Immediate
ADDA  ;ADD Address
ADDQ  ;ADD Quick
```

Nu grubler du måske over, hvorfor man skal have så mange (og lidt flere) kommandoer bare for at lægge to tal sammen, så lad os se på de enkelte kommandoers restriktioner.
Syntaxen for ADD er som følger:

```
ADD.s s<ea>,Dn
ADD.s Dn,d<ea>
```

hvor .s er enten .B, .W eller .L
Læg mærke til at mindst een af operanderne skal være et data-register. Du kan benytte hele listen fra 1-12 på s<ea>-pladsen, hvorimod d<ea> er begrænset til 3,4,5,6,9,10,11,12. Første linie lægger <ea> til indholdet i et data-register, anden linie lægger indholdet af et data-register til <ea>.

```
ADDI.s #<data>,<ea>
```

Her er .s ligeledes til fri afbenyttelse; .B, .W eller .L
Som du kan se, er begrænsningen af ADDI, at source-data skal være en konstant (Immediate data), og <ea> kan være 1,3,4,5,6,9,10,11,12. Vi haster videre med:

```
ADDA.s <ea>,An
```

som adderer adresse-registeret An med <ea>. Når vi arbejder med adresser sættes automatisk den restriktion, at kun words og longs er tilladte (.s må ikke være .B)! Men så må du også proppe hvadsomhelst ned på <ea>'s plads.

```
ADDQ #<data>,<ea>
```

Denne kommando ligner meget ADDI instruktionen. Forskellen er, at den er hurtigere at udføre, men at #<data>-konstanten skal ligge mellem 1 og 8.
<ea> kan være 1,2,3,4,5,6,9,10,11,12.

Simpel aritmetik

Subtraktion

Når vi kan lægge tal sammen, er det også rart, at kunne trække dem fra hinanden igen. Lad os kort se på de tilsvarende subtraktions-instruktioner:

```
SUB   ;SUBtract binary
SUBI  ;SUBtract Immediate
SUBA  ;SUBtract Address
SUBQ  ;SUBtract Quick
```

Vi gør dette kort, da syntaxen for SUB kommandoerne er de samme som for de tilsvarende ADD kommandoer, og det samme gælder for bredderne, så dem ignorerer vi for denne gang. Altså:

```
SUB <ea>,Dn
SUB Dn,<ea>
SUBI #<data>,<ea>
SUBA <ea>,An
SUBQ #<data>,<ea>
```

Det eneste nye vi skal huske er, at selvom SUBA kan udføres på både words og longs, vil MC680x0 altid sørge for, at resultatet er et longword. Lad os tage et lille eksempel:

```
move.l #$20000,a0
suba.w #$220,a0
```

A0 indeholder nu tallet (adressen) $0001FDE0, altså selvom vi kun lavede et minus i word-bredde, er resultatet alligevel et long word.
Mere er der vist ikke at sige om SUB instruktionerne, så lad os tage en enkelt instruktion mere - en meget speciel en:

```
NEG.s <ea>  ;NEGate
```

som svarer til udtrykket '0 - <ea>'. Fortegnet bliver altså kort og godt vendt, så plus bliver minus og omvendt. .s er .B, .W eller .L, og <ea> er det meste af listen, nemlig 1,3,4,5,6,9,10,11,12.

Multiplikation

Så går vi i gang med gange-stykkerne. Her har vi kun to kommandoer at gøre godt med, nemlig:

```
MULU ;MULtiply Unsigned
MULS ;MULtiply Signed
```

Syntaxen er ens for de to:

```
MULU <ea>,Dn
MULS <ea>,Dn
```

Bredden er altid word (uword) for de to operander, og resultatet er altid et longword (ulong). På MC68020 og frem kan man dog også multiplicere longs, men det ligger uden for denne bogs rammer. Du kan ikke sætte et direkte adresseregister i stedet for <ea> (pkt. 2 i listen), men ellers er slaget frit - også med indirekte adressering.
Vi husker på, at MULU regner uden fortegn; både på operander og på resultatet, mens MULS tager højde for fortegnet alle tre steder. Samtidig er det måske rart at bemærke, at idet begge operander er af bredden word, og resultatet er af bredden long, vil der aldrig ske et overflow.

Division

Ligesom ved MUL kommandoerne skal vi se på to divisions-instruktioner:

```
DIVU ;DIVide Unsigned
DIVS ;DIVide Signed
```

Syntaxen er den samme som for multiplikation:

```
DIVU <ea>,Dn
DIVS <ea>,Dn
```

men de opererer anderledes. <ea> kan stadig ikke være et A-register, men skal igen være af bredden word (uword). Til gengæld skal D-registeret være af bredden long (ulong) - dette er meget vigtigt.
Resultatet ligger bagefter i word-delen af D-registeret, og den rest, der blev af divisionen, ligger i det højeste word i samme register. Vi kan altså ikke regne med flydende komma (komma-tal) direkte ved hjælp af disse kommandoer. I højniveau-sprog kalder man divisionsresten for 'Modulo'.

Simpel aritmetik

Nu kan man sige, at det er noget sjusk, at divisionsresten skal ligge og flyde i den øverste del af vores register, men vi har tidligere lært en kommando, der gør det nemt at få hentet den ned, hvor vi kan nå den:

Eksempel 11

```
        move.l  #$10000,D0
        moveq   #4,D4
        divu    D4,D0           ;$10000/4
        swap    D0
        tst.w   D0              ;Rest=0 ?
        beq.s   GikOp           ;Ja!
        <..>                    ;Program stump
        rts

GikOp:
        <..>                    ;Program stump
        rts
```

I modsætning til MUL kommandoerne kan der meget nemt ske overflow her. Gudskelov vil V (oVerflow) flaget blive sat i disse tilfælde, så man kan lave en betinget forgrening, f.eks. 'BVS Fejl', som hopper til en label 'Fejl', hvis V-flaget er sat.
En anden ting du skal huske på, er at det er strengt forbudt at dividere med nul - gør man det, kommer Guru'en farende!

Vi har tidligere set, hvordan man kan benytte de betingede forgreninger til at konstruere løkker med. I dette afsnit skal vi se på en kommando, der indeholder både en tæller og en betinget forgrening i samme kommando. Det er en meget brugt instruktion, der giver uanede muligheder til dem, der forstår at bruge den til fulde, så lad os få præsenteret den med det samme:

DBcc ;Decrement and Branch if cc

Og syntaxen:

DBcc Dn,<label>

Denne kommando virker som en betinget branch, der har indbygget tæller, så den maksimalt kører løkken Dn+1 gange. Hvis betingelsen (cc = Condition Code) er **falsk** og **Dn >= 0** hoppes der til <label> i programmet. Er betingelsen **sand** eller **Dn = -1**, er løkken færdig, og programmet fortsætter med næste kommando. Tælleren (Dn) bliver altså talt 'baglæns' fra den maksimale værdi og ned til -1. Det skal også lige bemærkes at tallet i Dn, skal være af bredden word! Ydermere skal du lægge mærke til, at kommandoen er en Branch (forgrenings) instruktion, og derfor er begrænset til hop på ± 32KB, men da man i en løkke jo hopper én vej (tilbage) i sit program, er det nok mere logisk at sige, at forgreningen højst må være 32KB.

Lad os tage et simpelt eksempel, der sammenligner to tekster:

Eksempel 12

```
        lea     Tekst1,a0               ;Peg på 1. tekst
        lea     Tekst2,a1               ;Peg på 2. tekst
        move.w  #Length-1,d0            ;Længde af tekst -1
Loop:   cmpm.b  (a0)+,(a1)+             ;Sammenlign karakterer
        dbne    d0,Loop                 ;Næste karakter hvis ens
        move.w  #Length-1,d1            ;Længde af tekst -1
        sub.w   d0,d1                   ;Antal ens karakterer
        rts                             ;Forlad program

Tekst1: dc.b    'Introduktion til Assembler.'
Tekst2: dc.b    'Introduktion til Maskinkode'
Length  EQU     *-Tekst2
```

Når teksterne ikke længere er ens eller hele teksten er sammenlignet, forlades løkken og antallet af ens karakterer beregnes i D1. Husk på at en DBcc løkke altid køres een gang mere end tallet i D-registeret, idet løkken først forlades når Dn = -1. Dette er klaret i programmet med at benytte 'Length-1' i stedet for selve længden af de to tekster.

Til sidst tager vi lige et special-tilfælde, nemlig:

```
DBRA Dn,<label>
DBF  Dn,<label>     eller
```

De to kommandoer er ens, men nogle assemblere kan bedst lide den ene skrivemåde, andre bedst den anden. DBRA står for 'Decrement and BRAnch', og DBF står for 'Decrement and Branch if False' - Den eneste måde at komme ud af sådan en løkke er altså hvis Dn = -1, eller sagt med andre ord: Jeg er totalt ligeglad med betingelsen, bare kør løkken det antal gange, jeg beder om.
Eksempelvis skal vi have fyldt en tabel ud med tallene 0-9, og det kunne gøres således:

```
         lea      TabelEnd,a0  ;Peg på Tabel slut
         moveq    #9,d0        ;Største værdi
Loop:    move.b   d0,-(a0)     ;Læg værdier baglæns ned i tabellen
         dbra     d0,Loop      ;Formindsk værdi og fortsæt
         rts                   ;Forlad program

Tabel:   ds.b     10           ;Plads til de ti tal
TabelEnd:                      ;Slut på tabel
```

Her lægges tallene til rette i tabellen baglæns, da vores løkke jo starter med at have den maksimale værdi, og efterhånden som løkken arbejder sig færdig, bevæger A0 sig mod starten af tabellen. Læser man herefter tabellen i den rigtige rækkefølge, vil tallene stå med 0 forrest og 9 bagest.

Vi har tidligere set, at vi kan bryde den sekventielle udførsel af et program, v.h.a. hop- og forgreningskommandoerne. Men der findes andre metoder, der kun bryder sekvensen midlertidigt, for derefter at vende tilbage hvor man kom fra og fortsætte programmet. Dette gør man ved hjælp af '**Subroutiner**', som betyder '**under-programmer**'. Man benytter dette, hvis man skal have lavet den samme operation flere gange forskellige steder i programmet. Til dette har vi en afart af både hop- og forgrenings-kommandoen:

```
JSR <ea>    ;Jump to SubRoutine
BSR <label> ;Branch to SubRoutine
```

Disse to kommandoer fungerer præcis som de tilsvarende JMP og BRA instruktioner, dog med en enkelt undtagelse. Når JSR eller BSR udføres, gemmes PC på stacken, og når processoren første gang støder på en RTS kommando, vil den hente den gamle værdi tilbage til PC. Programmet afsluttes altså først, når der ikke er flere retur-adresser på stacken. Det er derfor, vores 'Return' kommando kaldes 'Return from Subroutine', altså 'vend tilbage fra underprogrammet'.
Det er værd at bemærke, at disse to kommandoer ikke ødelægger nogen af registrene. Det gør, at man kan lave et generelt underprogram, der forventer, at de data det skal bruge ligger i nogle specifikke registre. Dermed kan den samme generelle rutine benyttes til mange ensartede behandlinger af forskellige data. Dette kunne for eksempel være en rutine, der udskrev den tekst, som A0 pegede på, eller måske én der finder det største af tallene i D0 og D6. Lad os tage et eksempel:

Eksempel 14

```
            moveq   #1,d0           ;Linie nr. 1
            lea     Tekst,a0        ;I denne tekst
            bsr     FindTekst       ;Få pointer
            <udskriv>               ;Evt. udskrivning
            moveq   #3,d0           ;Linie nr. 3
            lea     Tekst,a0        ;Iden samme tekst
            bsr     FindTekst       ;Få pointer
            <udskriv>
            rts                     ;Afslut program

FindTekst:                          ;Subroutine
;IND: D0=Nummer på linie
;     A0=Pointer på tekst
;UD : A0=Pointer på linie
;D0 ødelægges

Loop:       tst.b   (A0)+           ;Fundet et 0?
            bne.s   Loop            ;Nej, prøv igen
            dbra    D0,Loop         ;Næste linie
            rts                     ;Færdig med subroutine

Tekst:
            dc.b    0,'Tekst linie 0'
            dc.b    0,'Tekst linie 1'
            dc.b    0,'Tekst linie 2'
            dc.b    0,'Tekst linie 3'
            dc.b    0,'Tekst linie 4'
```

Pyh, det var et langt eksempel. Programmet består af tre dele: Et hoved-program (linie 01 til 09), et underprogram (linie 11 til 20) og en tekst (linie 22 til 27). Hoved-programmet er ufærdigt, idet selve udskrivningsrutinerne mangler, men det lægger nogle data i de registre, som underprogrammet behandler.
Vores subroutine kræver, at der ligger et linie-nummer i D0 (0 til 4) og at A0 peger på teksten. Herefter finder den frem til den rigtige linie, ved at lede efter de nuller, der adskiller dem. Dette står som kommentarer i programmet i linie 12 til 15.
Lad os køre programmet igennem. Linie 01 beder om linie 1 i teksten, og linie 02 lader A0 pege på den. Linie 03 kalder vores underprogram, så vi hopper ned til linie 11. Da linie 11 kun indeholder en label og linie 12 til 16 er kommentarer, der ikke kommer til at ligge i programmet, vil processoren opfatte linie 11 og linie 17 som samme sted i hukommelsen. Linie 17 undersøger om A0 peger på et 0, og rykker A0 til at pege på næste adresse. Linie 18 hopper til linie 17, hvis der ikke var noget 0. Når vi finder vores 0 kommer vi ned i linie 19, der trækker 1 fra tallet i D0, og hvis dette ikke er -1, hopper den til linie 17 og fortsætter. Når vi har fundet den rigtige linie i teksten, sørger linie 20 for, at vi springer tilbage til vores hoved-program i linie 04. Her kunne der være en udskrivningsrutine, men den må du indtil videre tænke dig til. Linie 05 beder om et nyt linie nummer og linie 06 fortæller, at det stadig er den samme tekst vi leder i. Linie 07 springer endnu en gang ned til underprogrammet, som returnerer en pointer til tekstlinien i A0 når der returneres. Herefter afsluttes programmet i linie 09.
Resultatet af programmet er, at A0 vil pege på linien 'Tekst linie 1' når man når til linie 04 i programmet, og 'Tekst linie 3' når man er kommet til linie 08 i programmet.

Der ligger en rutine på disken, der kan skrive en tekst ud i CLI'en. Prøv at indsætte dette eksempel i den rutine, så du kan se, om jeg taler sandt.

Indtil nu har vi kun set på data bredder på bytes, words og long words. Nu er det på tide at kravle helt ned på bit-plan. Vi har fire kommandoer til at lege med de enkelte bits, nemlig

```
BSET  ;test a Bit and SET
BCLR  ;test a Bit and CLeaR
BCHG  ;test a Bit and CHanGe
BTST  ;Bit TeST
```

som skulle dække alle muligheder, nemlig tænde en bit, slukke en bit, ændre en bit og teste en bit. Som du kan se, indeholder samtlige kommandoer en test af bit'en, hvilket resulterer i, at Z flaget påvirkes af bit'ens status før ændringen. Hvis bit'en er 0 vil Z flaget blive sat (Z=1).
Lad os kaste et blik på syntaxen:

```
BSET  Dn,<ea>
BSET  #<data>,<ea>
```

hvor <ea> er 1,3,4,5,6,9,10,11,12, men bredden af operationen er afhængig af, hvad der står som <ea>. Hvis <ea> er et D-register, virker operationen på et helt long word, men er <ea> et sted i hukommelsen, kan bredden kun være en byte. Man kan altså godt sætte bit 30, i et register, men ikke i en variabel:

Eksempel 15

```
01       lea     Tal,A0
02       moveq   #30,D0
03       moveq   #0,D1
04       bset    D0,D1        ;Sæt bit 30 i D1
05       bset    D0,(A0)      ;Ups!
06       bset    #20,D1       ;Sæt bit 20 i D1
07       bset    #20,(A0)     ;Ups!
08       rts
09
10 Tal:  dc.l    0
```

Din assembler kan sagtens udføre dette lille program, men hvad vil der ske, når linierne 05 og 07 bliver udført? I begge tilfælde skal vi ned og rode i hukommelsen, og dermed kan vi kun benytte en bredde på en byte, men vi beder om bredder på et long word.
Processoren vil i dette tilfælde blive ved at tælle rundt i den samme byte, indtil den når bit nummer 30. Da der er 8 bits i en byte, vil den altså tælle tre gange rundt i byten (og har dermed talt til 24) og sætter derefter bit 6. Dette vil svare til at 'Tal' nu indeholder %01000000 = $40 i øverste byte og nuller i resten. Dermed klarede processoren skærene i denne omgang, for det svarer netop til bit 30, hvis man læser hele long word'et.
Det går først galt i linie 07. Her vil processoren tælle 2 gange rundt i byten (og har så talt til 16), og derefter sættes bit 4. Læses hele long word'et 'Tal' herefter, er bit 4 og 6 sat i den øverste byte, og det svarer til et long word, der er $50000000. I linierne 04 og 06 sættes de samme bits i D1, som dermed kan bruges som kontrol af resultatet. D1 vil altså have tændt bit 20 og 30, og indeholder derfor tallet $40100000.
Den eneste konklusion der kan uddrages af dette er, at man kan være heldig at det går godt, men man kommer nok til at bruge lang tid til debugning, hvis man

skal have det til at køre perfekt. Derfor: Skal du sætte bits i et long word, så hent det først ind i et data-register, sæt så bits'ne og skriv registeret tilbage i RAM:

Eksempel 16

```
     lea     Tal,A0
     moveq   #30,D0
     move.l  (a0),D1
     bset    D0,D1
     bset    #20,D1
     move.l  D1,(A0)
     rts

Tal: dc.l    0
```

På samme måde kan man slette en bit:

```
BCLR  Dn,<ea>
BCLR  #<data>,<ea>
```

hvor syntaxen er præcis den samme som for BSET, bit'en slettes bare i stedet for at blive sat. Og jeg ville også gentage mig selv, hvis jeg skulle komme med en smøre om:

```
BCHG  Dn,<ea>
BCHG  #<data>,<ea>
```

```
BTST  Dn,<ea>
BTST  #<data>,<ea>
```

Lad os derfor tage endnu et lille eksempel:

Eksempel 17

```
Led:  bchg   #1,$bfe001   ;Skift lyset i PowerLED
Vent: btst   #6,$bfe001   ;Er venstre mus trykket?
      bne.s  Vent         ;Nej, vent
      bchg   #1,$bfe001   ;Skift LED tilbage
      rts
```

Dette program benytter sig af direkte tilgang af et hardware register. Ser du i listen over Amiga'ens hukommelse (kapitel 6), opdager du, at det drejer sig om første register i PIA A. 'PIA' står for 'Port Interface Adaptor', men kaldes lige så ofte for CIA, 'Complex Interface Adaptor', som er håndteret af et par 8520 chips, der bl.a. styrer seriel- og parallel portene. Jeg vil ikke komme mere ind på disse.

Prøv at køre programmet mens du holder øje med din Power LED. Når du starter programmet vil LED'en slukke, herefter vil programmet vente på, at du trykker på den venstre knap på musen, tænde LED'en igen og forlade programmet.

Vi skal lige se, hvordan man kan rykke rundt på bits'ne i et tal. Nu skal du ikke tro, at man kan blande dem rundt mellem hinanden som et sæt spillekort, men man kan rykke alle bits i et tal samlet mod højre eller venstre. Man benytter tegnene '<<' og '>>' for at angive hvilken retning, man rykker i, og bagefter noterer man antallet af gange, man rykker en position:

```
%00010101 << 2  = %01010100
```

Alle bits er rykket to pladser til venstre, men hvad gør det mon selve tallet? Tallet %00010101 = $15 = 21 og tallet %01010100 = $54 = 84. Det lader altså til, at vi har fået ganget vores tal med 4. Hvis du leger lidt med shifting, vil du se, at

```
X<<1 = X*2
X<<2 = X*4
X<<3 = X*8
osv.
```

Det er jo netop det grundlæggende ved de binære tal, og hvis du har glemt dette, må du hellere læse afsnittet om talsystemer igen. Tilsvarende vil en shifting mod højre betyde, at du dividerer tallet.

Lad os kaste os over syntaxen for shift kommandoerne:

```
1. LSd  Dx,Dy      ;Logical Shift <direction>
2. LSd  #<data>,Dy
3. LSd  <ea>
```

Det lille '**d**' står som nævnt for '**direction**', altså hvilken vej du vil shifte. Sæt 'L' ind for venstre (Left) og '**R**' ind for højre (Right). Source operanden (den til venstre) indeholder antallet af gange, du vil shifte destinationen, og nu vil linienumrene ved syntaxen komme os til hjælp, der er nemlig flere muligheder:

```
1. Data-register  - mulig shift 0-63 gange
2. Konstant       - mulig shift 1-8 gange
3. Hukommelse     - mulig shift 1 gang
```

Dette skulle gerne vise, at det ikke var en trykfejl, at der kun er én operand i 3. syntax - der er jo ikke grund til at angive hvor mange gange, man vil shifte, når der kun er én mulighed.

Hvad mon der sker, når man shifter en bit ud af tallet? Den tager en lille omvej til de evige bitmarker, for den forsvinder ikke bare i første omgang. Den bit der shiftes ud, vil havne i både C- og X-flaget, ligegyldigt hvilken vej du shifter. Derimod vil den bit, der shiftes ind i tallet, altid være 0.

Bit shifting og rulning

Der er en enden kommando, der gør stort set det samme, nemlig

```
1. ASd Dx,Dy      ;Arithmetic Shift <direction>
2. ASd #<data>,Dy
3. ASd <ea>
```

hvor syntaxen er den samme som for det Logiske Shift. Som navnet antyder, benyttes denne afart af shiftet til aritmetiske operationer, hvorimod det Logiske Shift benyttes til alt det andet. Den eneste rigtige forskel er, at ved ASR kopieres MSB i stedet for at indsætte et 0. Dermed beholder tallet sit fortegn. Derudover er der en lille forskel i måden oVerflow flaget sættes, men det er så sjældent benyttet, at vi vil glemme det igen.

Det skal også lige bemærkes, at når shifting benyttes som multiplikation og division, er det en pseudo-operation, f.eks. tallet 1 shiftet en gang til højre ikke give 1/2, men 0. Men lad os da tage et lille eksempel. Vi så i kapitlet om direktiverne, at man kan lave to konstanter, hvis man skal bruge både antallet af pixels og bytes i bredden af en skærm. Dette kunne gøres ved hjælp af kun én konstant, idet der er 8 pixels (bits) i en byte. Vi kunne derfor finde antallet af bytes, på følgende måde:

Eksempel 18

```
        move.w    #Bredde,d0    ;Bredden i pixels
        asr.w     #3,d0         ;Divider med 8
        rts
```

Et andet eksempel kunne være, at man vil benytte den indirekte forskydning i en tabel af long words:

Eksempel 19

```
01        moveq     #2,D0         ;Vi vil have tal nummer 3
02        lea       Tabel,A0      ;I denne tabel
03        asl.w     #2,D0         ;*4 = bytes -> longs
04        move.l    (A0,D0),D1    ;Hent tallet
05        rts
06 Tabel:
07        dc.l      0,1,2,3,4,5
```

Hvis vi ikke konverterer D0 til et long offset i linie 03, vil vi jo få den tredie byte i tabellen, som ligger midt inde i det første long word. Køres programmet som det er, vil D1 indeholde tallet $00000002, men hvis linie 03 udelades, vil tallet i D1 være $00000000. Og vi havde virkelig fået problemer, hvis vi havde bedt om tal nummer 3 i tabellen, for så havde vi forsøgt at hente et long word fra en ulige adresse, og Guru'en var kommet susende!

Der er flere af denne type kommandoer, hvor de kommer fra, men nu skal vi se på bit rulning eller rotering. Syntaxen er der ikke noget nyt i:

```
1. ROd Dx,Dy     ;ROtate <direction>
2. ROd #<data>,Dy
3. ROd <ea>
```

Men der er igen et par forskelle, vi heller må kigge på. For det første vil den bit, der bliver puffet ud af tallet, tage en grusom hævn og dukke op igen i den anden ende, som den bit, der kommer ind i tallet. For det andet vil den bit, der ruller ud kun sætte sig i C flaget. Vi smider altså ikke bits væk, idet det der bliver smidt ud i den ene ende, kommer ind i den anden, og samtidig vil samme bit dukke op i C flaget.

Lad os se skematisk på det:

```
%00001111 >> 2 = %11000011 og C=1
%00001111 << 2 = %00111100 og C=0
```

Har vi et word, hvor vi ønsker at bringe de fire laveste bits op i de fire højeste, kan man gøre det på forskellige måder:

Eksempel 20

```
move.w    #$000F,D0    ;Tallet
lsl.w     #8,D0        ;Ryk 8 pladser
lsl.w     #4,D0        ;Ryk resterende 4
rts
```

Det var ikke så smart.

```
move.w    #$000F,D0    ;Tallet
moveq     #12,D1       ;Shift værdi
lsl.w     D1,D0        ;Shift 12 pladser
rts
```

Det var bedre, men ødelægger D1.

```
move.w    #$000F,D0    ;Tallet
ror.w     #4,D0        ;Rul på plads
rts
```

Juhu, slet ikke så dårligt! En sølle kommando var nok, og vi ødelægger ikke andre registre. Derimod får vi C flaget sat, men det er sjældent, det har nogen betydning.

Logiske operationer

I sidste kapitel så vi, hvordan man kan påvirke en enkelt bit v.h.a. bit-instruktionerne. I dette kapitel holder vi os til bits'ne, men vi nøjes ikke nøvendigvis med at påvirke én ad gangen. Til dette formål har vi fire logiske operationer - hvis du kender noget til digital elektronik, er du med garanti stødt på disse før:

AND, OR, EOR og NOT

De tre første instruktioner benytter både et source og en destination, mens NOT kun opererer på én operand.

Det Logiske AND

Et logisk AND tænder kun de bits i destinationen, hvor bit'en i forvejen er tændt i både source og destination, og slukker alle andre:

Source : 0101
Dest : 0011
Resultat : 0001

Som du kan se, har vi nu fået slukket flere bits, end vi har fået tændt, og det er som regel dette, man benytter AND til; altså til at slukke bits man ikke ønsker skal have indflydelse på en kommende kommando. Lad os lige snuppe syntaxen for AND:

```
1. AND.s  <ea>,Dn       ;Logical AND
2. AND.s  Dn,<ea>       ;Logical AND
3. ANDI.s #data,<ea>    ;AND Immediate
```

hvor .s kan være både .B, .W og .L og for <ea> er der lidt blandede bolcher for hver af de tre syntax-linier:

<ea> i 1: 1,3,4,5,6,7,8,9,10,11,12
<ea> i 2: 3,4,5,6,9,10,11,12
<ea> i 3: 1,3,4,5,6,9,10,11,12

Lad os se på et eksempel. Forestil dig, du vil lave en rutine, der kan konvertere et tal til hex-format, så du kan skrive det ud på skærmen. Dette kan man f.eks. gøre således:

Eksempel 21

```
01      moveq         #0,D0               ;Slet hele registeret
02      move.b        #tal,D0             ;Tallet du vil vise
03      move.w        D0,D1               ;Lav en kopi
04      and.b         #$0f,D1             ;Nulstil bit 4-7
05      lea           HexTal,A0           ;Peg på liste over ASCII hex-tal
06      lea           Asc_End,A1          ;Peg på slutning af resultat
07      move.b        (A0,D1.w),-(A1)     ;Sæt 'mindste' ciffer
08      lsr.b #4,D0                       ;Hent bit 4-7 til bit 0-3
09      move.b        (A0,D0.w),-(A1)     ;Sæt 'højeste' ciffer
10      rts                               ;Afslut program

11 HexTal:
12      dc.b          '0123456789ABCDEF'
13 Resultat:
14      dc.b          '$00'
15 Asc_End:
```

Du kan nu vise tallet på skærmen ved at udskrive de tre karakterer, som 'Resultat' peger på (linie 13 og 14). Prøv selv at lave en rutine, der kan udskrive et long word. Der er et løsningsforslag på disken, der følger med bogen.

Det logiske ANDI kan direkte benyttes til at slukke flag i CCR eller SR. Dette gøres simpelthen ved f.eks:

```
ANDI.B #%01111,CCR ;Sluk X flaget (bit 4)
```

Jeg vil ikke komme ind på behandlingen af SR, da det som nævnt ligger udenfor denne bogs omfang. Roder du selv med det, skal du være opmærksom på, at al påvirkning af SR kun må ske i Supervisor Mode, og da jeg ikke vil komme ind på dette, synes jeg vi alle skal glemme det for et stykke tid.

Det Inclusive Logiske OR

Det Logiske OR benyttes, modsat AND, til at tænde for bits. Syntaxen er den samme som for AND, så lad os springe direkte til eksemplerne:

```
Source  : 0101
Dest    : 0011
Resultat: 0111
```

Kun når begge bits er nul, giver resultatet 0. Og syntaxen:

```
1. OR.s <ea>,Dn      ;Logical OR
2. OR.s Dn,<ea>      ;Logical OR
3. ORI.s #data,<ea>  ;OR Immediate
```

Hvad kan man så bruge det til? Lad os tænke os, at man har fået opgivet styrken af de tre nuancer rød, grøn og blå, og skal bruge den samlede farve-værdi til f.eks. Copper-listen (Hardware-programmering). På Amiga'en lagres en farve som et word med følgende bit-konfigurering:

```
Farve: 0000 RRRR GGGG BBBB
```

altså bit 0-3 er styrken af blå, bit 4-7 er styrken af grøn og bit 8-11 er styrken af rød (da styrken beskrives af 4 bits har vi altså 16*16*16 = 4096 farver). Bit 12-15 er ubenyttede, og skal sættes til 0. Hvis vi, som eksempel, siger, at D1=$0008 (styrken af rød), D2=$000A (styrken af grøn) og D3=$000F (styrken af blå), og resultatet, altså den samlede farve-værdi, skal ligge i D0, kan man gøre følgende:

Eksempel 22

```
1 Color: moveq    #0,D0     ;Slet D0 (resultat)
2        move.b   D3,D0     ;Indsæt blå nuance
3        lsl.w    #4,D2     ;Læg grøn nuance på bit 4-7
4        or.w     D2,D0     ;Indsæt grøn nuance
5        lsl.w    #8,D1     ;Læg rød nuance på bit 8-11
6        or.w     D1,D0     ;Indsæt rød nuance
7        rts                ;Afslut rutine
```

Hvis vi følger dette lille program linie for linie, får vi:

Linie	D0	D1	D2	D3
1	$0000	$0008	$000A	$000F
2	$00<u>0F</u>	$0008	$000A	$000F
3	$000F	$0008	$00<u>A0</u>	$000F
4	$00<u>AF</u>	$0008	$00A0	$000F
5	$00AF	$0<u>800</u>	$00A0	$000F
6	$0<u>8AF</u>	$0800	$00A0	$000F

Resultatet er altså at D0=$08AF, som er den samlede værdi af farven.

Også OR kan benyttes til at ændre flag i CCR og SR, f.eks.:

```
ORI.B #%01000,CCR ;Tænd for N flaget (bit 3)
```

Det Logiske Exclusive OR, EOR

Så er turen kommet til EOR, og igen er syntaxen den samme som for AND:

```
1. EOR.s <ea>,Dn      ;Logical EOR
2. EOR.s Dn,<ea>      ;Logical EOR
3. EORI.s #data,<ea>  ;EOR Immediate
```

så lad os istedet se, hvad kommandoen kan gøre:

```
Source  : 0101
Dest    : 0011
Resultat: 0110
```

altså ligesom OR med den undtagelse, at der kun tændes en bit, hvis source og destination er forskellige, og den slukkes, hvis begge var tændt fra starten. Dette kan måske afgøre, om en program-del eller subrutine skal udføres eller ej. Hvis 'tændt' =1 og 'slukket' =0 kan man, hver gang man 'trykker' på kontakten, gøre:

Eksempel 23A

```
         eor.b        #1,Kontakt
         <...>
Kontakt: dc.b         0
```

Hver gang EOR-kommandoen udføres, ændres byten fra slukket til tændt og omvendt. Herefter kan man f.eks. benytte sig af et TST.B til at afgøre, om der skal ske noget eller ej:

Eksempel 23B

```
           tst.b      Kontakt        ;Test kontakten
           beq        Ingenting      ;Hop, hvis slukket
           <...>                     ;Udfør rutine
Ingenting:
           rts                       ;Forlad rutine
```

Den Logiske Complement, NOT

Den sidste af de logiske operationer skal kun have én operand. Syntaxen er:

```
NOT.s <ea>
```

hvor .s kan være .B, .W og .L og <ea> kan være 1,3,4,5,6,9,10,11,12.

Logiske operationer

NOT laver en bitmæssig invertering af operanden, dvs. alle tændte bit (1) bliver slukkede (0), og omvendt:

```
Dest       0101
NOT(dest)  1010
```

Lad os tage et mere eller mindre søgt eksempel på brugen af denne kommando. Forestil dig, vi har en betinget løkke af typen DBcc, og ønsker at vide, om vi forlod løkken, fordi tælleren var -1, eller om betingelsen var sand. Vi kunne gøre dette på følgende måde:

Eksempel 24

```
        <...>                 ;Tilfældig kode
Loop:   <...>                 ;Kode i løkken
        dbeq    d0,Loop       ;Hop hvis D0<>-1 eller
                              ; betingelsen var falsk
        not     d0            ;Inverter D0 (-1 = 0)
        beq     Nul
        <...>                 ;Betingelsen sand
Nul:    <...>                 ;Løkken sluttede
```

Her benytter vi, at d0 = -1 hvis løkken er kørt igennem til den bitre ende, mens d0 >=0 hvis betingelse har stoppet løkken. Det er et tamt eksempel, men du må leve med det, for NOT bliver ikke benyttet mere i denne bog.

Inden du går i gang med at skrive lange programmer, er der lige et par småting, du bør vide om CLI'ens opførsel. For det første holder CLI selv øje med programmers fejl-meddelelser. Dette gør den ved at undersøge tallet, der står i D0, når programmet forlades. Er dette tal 0, er CLI glad og gør ikke det mindste, der kan gøre livet surt for dig. Er tallet derimod ikke 0, må du regne med, at du får smidt noget i nakken. Den rigtige måde at afslutte et program på, er derfor:

Eksempel 25

```
        ....                ;Dit program
        clr.l    d0         ;Slet D0
        rts                 ;Tilbage til CLI

eller benyt

        moveq    #0,d0      ;Slet D0
```

i stedet for 'clr.l d0'. Der er et par retur-værdier, som har en defineret betydning:

```
D0 = 5     ;Warn
D0 = 20    ;Fatal fejl
```

disse retur-værdier kan benyttes, hvis du vil bruge dit program i et script, f.eks. i din Startup-Sequence. Hvis du returnerer en 'fatal fejl' meddelelse, vil CLI ikke undlade at gøre dig opmærksom på det, så vær forsigtig her. Derimod sker der ikke noget ved at returnere et 'Warn'. Hvis du f.eks. vil lave en Startup-Sequence, der laver én ting hvis du trykker på musen, og noget andet hvis du ikke gør, laver du et lille program, der sætter undersøger om knappen er trykket, og sætter et Warn, hvis den er. Derefter kører du programmet i din Startup-Sequence:

Eksempel 26

```
CheckMouse      ;Dit program
IF WARN         ;Er knappen trykket?
Execute User1   ;Ja, udfør script 'User1'
EndCLI          ;Afslut
ENDIF
Execute User2   ;Ellers spring til 'User2'
EndCLI          ;Afslut
```

Det er ikke så svært vel? Lad os kigge på et andet gimmick ved CLI. Når du starter et program fra CLI, får du nogle små nyttige værdier:

```
A0 = Pointer til argumenter
D0 = Længde af argument-streng
```

Et par facts om CLI

Hvis du har startet dit program med:

```
CheckMouse -h
```

for at angive at er skal komme et Warn, hvis højre knap er trykket, vil A0 pege på det sted i hukommelsen, hvor argument-strengen, '-h', starter. Læser du derefter indholdet af A0, vil tegnet '-' stå der. Længden af strengen er som sagt givet i D0, men pas på, for det er længden af hele strengen inklusiv <Return>, længden vil derfor være 3 i dette tilfælde. Det smarteste vil derfor være at starte med følgende linier i dit program, inden du undersøger argumenterne:

Eksempel 27

```
        subq    #1,d0
        move.b  #0,(a0,d0)          ;Slet <Return>
```

herefter kan du undersøge argumenterne indtil du støder på et 0 eller til du har læst det antal karakterer, der står i D0.

Nu har jeg jo fået skrevet helt om til omslaget, så du har vel gættet, at der ikke kommer mere i denne bog. Godt nok har jeg ikke gennemgået hele instruktionssættet, men dog så meget, at du burde være i stand til at lave programmer selv. Der medfølger en mindre bunke eksempler, du frit kan bruge som inspiration eller i egne programmer.

Glad for at skrive, fuld af gode idéer!?!
Står du og tripper for at udgive dine egne bøger, har vi løsningen til dig.
Har du en god idé til en bog, enten faglitterært eller skønlitterært, giver vi dig mulighed for at kunne virkeliggøre dine drømme.

Alle idéer er velkomne!

Skriv til:
Dansk Medie Hus ApS
Hillerødgade 81 - 83
2200 København N

Mrk. Bogidé

DANSK MEDIE HUS ApS
FORLAG · MARKETING

Hillerødgade 81 - 83 • 2200 København N
Tlf: +45 3833 1011 • +45 3833 3134

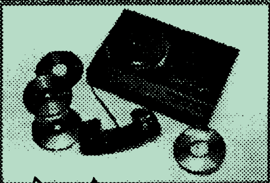
COMPUTER BUSINESS CENTER

BETAFON ApS

Istedgade 79 & 110 • 1650 København V • Tlf.: 3131 0273 • Fax: 3131 1498

De nye stærke 32 bit Amiga maskiner

Giver dig: 2 MB Ram • Over 262000 farver af et udvalg på 16.8 millioner • Kickstart 3.x



CD32

• incl. 2 spil & fakta leksikon

Kr. 3495,-

3495,-
Excl. Skærm

• 5 System disks
• Mus

Amiga 4000

• 120 MB IDE-AT harddisk
• Mus & tastatur
• excl. skærm

A 4000/30
• 4 MB Ram ialt
Pris 12250,-

A 4000/40
• 6 MB Ram ialt
Pris 18995,-

Postordre salg

Varebestilling inden kl.14 sendes samme dag.

☎ 3131 0273

Ring for prisliste til:

- ❑ Amiga
- ❑ PC
- ❑ Amiga spil
- ❑ PC spil
- ❑ CDTV

Alle priser er incl. moms og gældende til den 31/12-93
Der tages forbehold for trykfejl, pris- & specifikationsændringer samt udsolgte varer

This was brought to you

from the archives of

http://retro-commodore.eu